MASTER IN COMPUTER SCIENCE

Parallel Mercury

Tannier, Jérôme

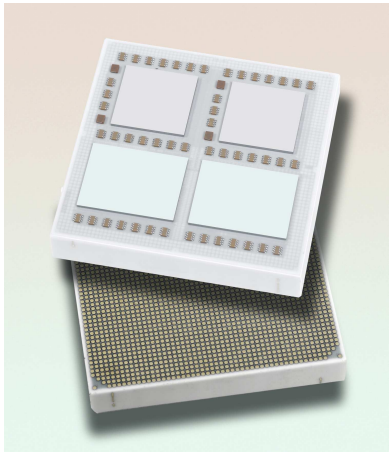
Award date:
2007

Awarding institution:
University of Namur

Institut d'informatique
Facultés Universitaires Notre-Dame de la Paix
21, rue Grandgagnage
B-5000 Namur
Belgium

Parallel Mercury

Jérôme Tannier



Thesis submitted for the degree of
Master of Computer Science in the University of Namur

June, 2007

Abstract

Mercury is a relatively new declarative programming language aimed at resolving the issues commonly found in programming languages of that paradigm. This thesis presents the improvements which have recently been implemented in the Mercury compiler regarding parallelism. The document is divided as follows. To begin with, we define a few notions related to the imperative and declarative paradigms. Then, we introduce the Mercury language mainly through its syntax, goals, types, modes, and determinism. Afterwards, we pay interest in parallelism in Mercury by presenting what has been done so far in the Mercury compiler. Next, we introduce the work which has recently been done to the Mercury compiler namely implicit parallelism and granularity control. Before we conclude this thesis, a parallelism implementation comparison is presented with the two main other declarative programming languages namely Haskell and Prolog.

Résumé

Mercury est un langage de programmation déclaratif relativement récent ayant pour but de résoudre les problèmes couramment rencontrés dans les langages de programmation de ce paradigme. Ce mémoire présente les améliorations qui ont été récemment implémentées dans le compilateur Mercury concernant le parallélisme. Ce document est structuré comme suit. Pour commencer, nous définissons un certain nombre de notions ayant trait aux paradigmes impératifs et déclaratifs. Ensuite, nous introduisons le langage Mercury principalement à travers sa syntaxe, ses buts, ses types, ses modes et son déterminisme. Par la suite, nous nous intéressons au parallélisme dans Mercury en présentant ce qui a été implémenté jusqu'à présent dans le compilateur Mercury. Puis, nous introduisons le travail qui a récemment été fait dans le compilateur Mercury à savoir le parallélisme implicite et le contrôle de granularité. Avant de conclure ce mémoire, une comparaison sur l'implémentation du parallélisme est présentée avec les deux autres principaux langages de programmation déclaratifs à savoir Haskell et Prolog.

Acknowledgements

Working on the Mercury compiler has been a pleasure. I thank my supervisors Mr Wim Vanhoof and Mr Zoltan Somogyi for making this research possible. Thanks are also due to Julien Reid who did a great job at doing all the paperwork in order to get my visa.

I thank the Mercury team that is Julien Fisher, Ralph Becket, Peter Ross, Mark Brown and Ian MacLarty. They helped me a lot during my internship at the University of Melbourne especially at the beginning when everything looked so complicated. Without their help, I would have never achieved the understanding of what I have now of the Mercury compiler.

I would like to thank also Meera, Daniel, Anna, Dawn, Kristy, and Sarah, my closest aussie friends who I will never forget. Thank you guys for your support when I was in Australia. It has been a pleasure meeting you! I am so much looking forward to seeing you again.

The biggest thank you is due to my parents for making this stay in Australia possible and therefore working at the University of Melbourne. Without them, I would have never accomplished that dream. I will never forget what they did.

Jérôme Tannier

Namur, May 2007

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Declarative programming languages have been around for more than thirty years [2]. A program is declarative if it describes the characteristics of a problem rather than its solution i.e. the algorithm. Declarative programming usually refers to functional and logic programming and is in contrast with imperative programming which requires the programmer to specify explicitly the manipulation of the state of the computer system [3].

Functional programming is a paradigm based on functions. A program consists of a set of functions. The execution of a functional program is the evaluation of a mathematical expression. A functional program always delivers the same output for a given input and, therefore, contains no side-effects. In other words, a functional program is said to be deterministic [4]. Haskell is one of the many functional programming languages.

Logic programming is a paradigm based on logic axioms. In a logic program, axioms represent the knowledge of the domain of a problem and the assumptions needed to solve that problem. A logic program is executed like a mathematical proof. The output of a logic program is computed with the use of deduction and the derivation of logical consequences [5]. The most well-known logic programming language is Prolog.

The declarative programming paradigm is an alternative to the imperative counterpart. The latter is known for its many issues. Firstly, writing a correct algorithm is not easy. Secondly, understanding an algorithm is difficult for developers who have not implemented it. Finally, verifying an algorithm correctness consumes a significant development effort [6]. However, despite their advantages over imperative programming languages, declarative programming languages have not managed to gain widespread use. Their infrequent use results from a few factors: lack of collegiate training in declarative languages, uneasy syntaxes of some languages, inefficient compilers and run-times, and restricted domains of applicability [6].

Functional and logic programming languages are programming languages that join in a single paradigm the features of functional programming and logic programming which complement to each other [7]. Mercury is one of these functional and logic programming languages. It is being developed at the Department of Computer Science and Software Engineering at the University of Melbourne in Australia. Mercury is aimed at resolving the issues commonly found in declarative programming languages. Mercury's type, mode and determinism systems allow a large percentage of program errors to be detected at compile-time. The Mercury compiler is highly optimized and delivers efficiency close to the one of imperative programming languages.

The Mercury compiler is distributed under the GPL license. The supported platforms are Microsoft Windows, MacOS X and various Unix-like operating systems (Linux, *BSD, Solaris, ...). The first version was released on April 8th, 1995. Even though Mercury has not reached the 1.0 version, it is considered ready for production use. It is being used in two software development companies namely "Mission Critical IT"(`http://www.missioncriticalit.com`) and "Logical Types, LLC"(`http://www.logicaltypes.com/`). The actual reason why Mercury is not at version 1.0 is that the compiler does not fully implement the reference manual. The unimplemented features are not crucial to developing applications though.

Chapter 2

Mercury

2.1 Mercury by example

A logic program is a representation of a domain for which we have to solve a problem. It contains a set of logic axioms or atoms and make use of terms representing the objects of the problem.

Terms are defined in the following way [8]:

- a variable is a term
- a constant is a term
- if $t_1, ..., t_n$ are terms
  if f/n is a function of arity n
  then $f(t_1, ..., t_n)$ is a term

Atoms are defined in the following way [8]:

if p/n is a predicate of arity n

if $t_1, ..., t_n$ are terms

then $f(t_1, ..., t_n)$ is an atom

In a logic program, predicates are defined using rules i.e. characteristics of the domain, and facts i.e. assumptions on the domain. The following is a simplified example which will eventually be turned into a Mercury program.

Example 2.1.1 *Simplified ancestor and parent predicates*

```
parent("Mathieu","Francois").
parent("Mathieu","Angele").
parent("Angele","Jean").

ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
ancestor(X,Y) :- parent(X,Y).
```

parent and *ancestor* are predicates. There are a number of facts in this example. Mathieu has two parents: Francois and Angele. Moreover, one of Angele's parents is Jean. Mathieu, Francois, Angele and Jean are terms. The last two lines of the example are rules. Indeed, Y is an ancestor of X if either Z is a parent of X and Y is an ancestor of Z, or Y is a parent of X.

We would like to know who are Mathieu's ancestors. That problem can be translated into the following logic query:

```
?- ancestor("Mathieu",Y).
```

The execution of a logic program is like a theorem proof. It accounts for a series of deductions i.e. a derivation built upon logic axioms. This can be formalized in an SLD-tree which represents all the solutions that can be built for a given problem [8]. The root of the tree is the query. A node in the SLD-tree represents the state of a derivation. A node has one or several child nodes. Each one corresponds to the unification of the parent node with a clause. A unification is a pattern matching operation between two terms. Two terms unify if there is a substitution that makes the terms identical [9]. A path in the SLD-tree is a derivation, and, therefore, a solution to the problem.

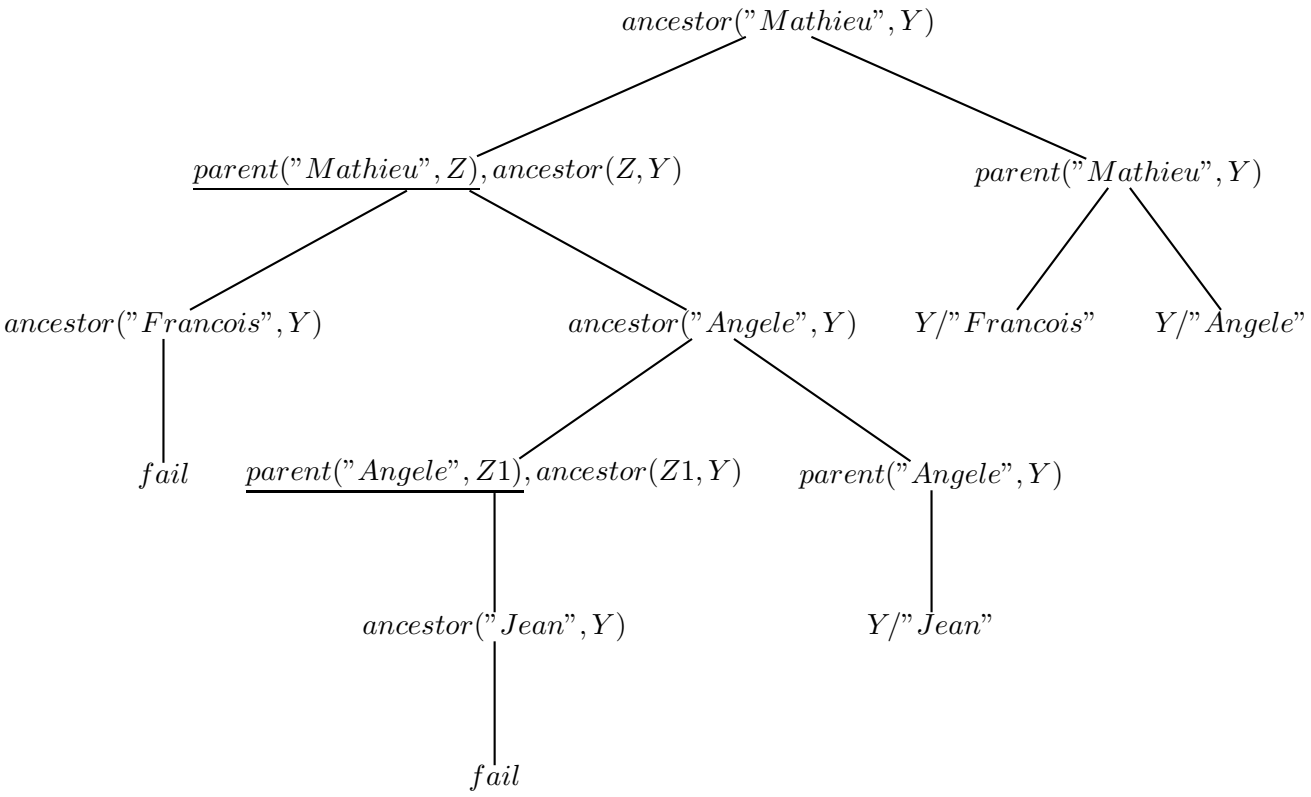


Figure 2.1: SLD tree from Example 2.1.1

In Figure 2.1, we use a depth-first search and backtrack mechanism with a leftmost atom selection to solve the query. In other words, we try to search paths to their completion before trying new ones. When a path fails, the algorithm backtracks to the last point at which it

could have chosen a different clause for unification, and then tries an unexplored alternative [10]. Also, if a node is composed of several atoms, we need to select which one is to be considered for unification. In the example, the atom selected is the one underlined i.e. the node on the left-hand side. However, the atom selection is up to the logic language and so is the selection of the clause which we unify an atom to.

The resolution of the problem goes as follows. The two rules in Example 2.1.1 can be used to resolve *ancestor("Mathieu",Y)*. Therefore, the root of Figure 2.1 has two child nodes. The first one results from the selection of *ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y)*. *Mathieu* unifies with X. What we need to resolve is now: *parent("Mathieu",Z),ancestor(Z, Y)*. We decide to consider the first atom for unification. *parent("Mathieu",Z)* unifies with the first two facts of Example 2.1.1. Therefore, the node *parent("Mathieu",Z), ancestor(Z,Y)* has two child nodes. Let us consider the unification with *parent("Mathieu","Francois")*. The resulting node is *parent("Mathieu","Francois"), ancestor("Francois",Z)*. As *parent("Mathieu","Francois")* is true, the second atom i.e. *ancestor("Francois",Z)* only matters for solving the problem. However, *ancestor("Francois",Z)* fails to unify as Francois has no parents. Once all the possibilities i.e. branches of the tree are built, we have the answer to our query. Mathieu's ancestors are Jean, Francois and Angele.

The following is Example 2.1.1 written in Mercury.

Example 2.1.2 *ancestor and parent predicates written in Mercury*

```
:- module ancestor.
:- interface.
:- import_module io.

:- pred main(io::di,io::uo) is det.

:- implementation.
:- import_module string.

main(!IO) :-
    ancestor("Mathieu",Y),
    io.write_string(Y,!IO),
    io.nl(!IO).

:- pred parent(string,string).
:- mode parent(in,out) is nondet.
:- mode parent(out,in) is nondet.

parent("Mathieu","Francois").
parent("Mathieu","Angele").
parent("Angele","Jean").

:- pred ancestor(string,string).
:- mode ancestor(in,out) is nondet.
```

```
:- mode ancestor(out,in) is nondet.

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
ancestor(X,Y) :- parent(X,Y).
```

We analyse in the next section what this Mercury program is actually made of.

2.2 Syntax

A Mercury program is a set of modules. Each item in a Mercury module is either a declaration or a clause [11]. A declaration is an item beginning with a :- symbol.

Example 2.2.1 *Declarations*

```
:- module ancestor.
:- interface.
:- import_module io.
:- pred main(io::di,io::uo) is det.
:- implementation.
:- import_module string.

:- module ancestor.
```

The module name must be the same as the filename.

```
:- interface.
```

This is the interface declaration. Everything contained in this section is visible to the outside world i.e. the other modules.

```
:- import_module io.
```

These are the imported modules in the interface. Every module imported should be used by the predicates defined in the interface. In this case, the module *io* is used by the predicate *main*.

```
:- pred main(io::di,io::uo) is det.
```

This is the predicate *main* which is visible to the other modules since it is declared in the interface section. *main* has two arguments of the type *io*. Every predicate that performs I/O has to have an *io* type input argument describing the state of the world at the time the predicate is called and an *io* type output argument describing the state of the world after the call. This is how Mercury allows programs to communicate with the outside world without compromising its mathematical integrity. We will look into details later what *di*, *uo* (mode) and *det* (determinism) mean.

```
:- implementation.
```

This is the implementation declaration. Everything contained in this section is not visible to the outside world. The predicates declared in the interface are implemented in here. Other predicates can be declared and are implemented in this section.

Predicates are defined by clauses which are either facts or rules. A Horn clause is of the form $H :- B_1, ..., B_n$ where $H, B_1, ..., B_n$ are atoms. H is the head of the clause. $B_1, ..., B_n$ is the body of the clause. A fact is a clause without a body. A rule is a clause with a non-empty body.

Example 2.2.2 *Facts*

```
parent("Mathieu","Francois").
parent("Mathieu","Angele").
parent("Angele","Jean").
```

Example 2.2.3 *Rules*

```
ancestor(X, Y) :- parent(X,Z),ancestor(Z,Y).
ancestor(X, Y) :- parent(X,Y).
```

As said previously, Mercury is not only a logic programming language but also functional. A function in Mercury is basically a predicate with a return value. *parent* and *ancestor* predicates can be rewritten as functions.

Example 2.2.4 *Functions*

```
:- func parent(string) = string.
:- func ancestor(string) = string.
```

2.3 Goals

A goal is the right-side of a clause i.e. the body of a rule [11]. The most common kind of goals are [12, 13]:

$lhs - term = rhs - term$ **:** a unification. It can either be a construction, deconstruction, assignment or equality test.

- $X = data\_constructor(Y_1,\ ...\ Y_n)$ is a construction if initially $\forall\ i\ Y_i$ is instantiated and X is not.
- $X = data\_constructor(Y_1,\ ...\ Y_n)$ is a deconstruction if initially X is instantiated and $\forall\ i\ Y_i$ is not.
- $X = Y$ is an assignment if initially either X or Y is instantiated.
- $X = Y$ is an equality test if X and Y are instantiated.

$p(arg_1,\ arg_2,\ ...,\ arg_n)$ **:** a call to a procedure or a function. In case of a function, there is a return value which can be stored using a unification like in the following:

$X = f(arg_1,\ arg_2,\ ...,\ arg_n)$.

Goal, *Goal*2 **:** a conjunction of goals. A subgoal is called a conjunct. A conjunction succeeds if all of its conjuncts succeed. A conjunction can be plain or parallel. A plain conjunction has its conjuncts executed sequentially. A parallel conjunction has its conjuncts executed in parallel and they might have variables in common (see Section 3).

Goal; *Goal* **:** a disjunction of goals. A subgoal is called a disjunct. A disjunction succeeds if any of the disjuncts succeeds. A disjunction is a switch if each disjunct has a unification that tests the same instantiated variable.

Example 2.3.1 *A switch on L*

```
(
    L = [], foo(...)
;
    L = [H|T], foo(...)
)
```

In Example 2.3.1, if the variable L is instantiated then the disjunction is a switch.

$\backslash +$ *Goal* **:** a negated goal. A negation succeeds if the negated goal fails and vice-versa. It is equivalent to '**if** *Goal* **then fail else true**'.

if *CondGoal* **then** *ThenGoal* **else** *ElseGoal* **:** if *CondGoal* succeeds then *ThenGoal* is executed, *ElseGoal* otherwise. It is equivalent to '$CondGoal -> ThenGoal; ElseGoal$'.

some $Vars$ *Goal* **:** an existential quantification. The variables $Vars$ are local to *Goal*. A variable appearing outside the quantification with the same name as one of the quantified variables is considered as a different variable.

all $Vars$ *Goal* **:** a universal quantification. It is equivalent to '**not** (**some** $Vars$ **not** *Goal*)'.

Variables that are not explicitly quantified are implicitly quantified. Head variables are universally quantified. Other variables are implicitly existentially quantified around their closest enclosing scope [14]. For instance, consider the following example.

```
...
A = foo(X,Y,Z),
( A = ..., B = 4 ->
    C = B + ...
;
    C = B + ...
),
...
```

We make an assumption that the variable B does not appear after the "if then else" goal. B is instantiated in the condition goal and used in the "then" and "else" parts of the "if then else" goal. In that case, B is implicitly existentially quantified inside the "if then else" goal. More on this can be found in [13].

A very common syntactic sugar are the state variables. A state variable is written '!.X' or '!:X', denoting the current or next value of the sequence labelled X. An argument '!X' is shorthand for two state variable arguments '!.X, !:X'; that is, 'p(..., !X, ...)' is parsed as 'p(..., !.X, !:X, ...)' [13].

Example 2.3.2 *Fibonacci written without using state variables*

```
:- pred fibonacci(int::in,int::out) is semidet.
fibonacci(X, Y) :-
    (
        X = 0, Y = 0
    ;
        X = 1, Y = 1
    ;
        X > 1,
        J = X - 1,
        K = X - 2,
        fibonacci(J,Jout),
        fibonacci(K,Kout),
        Y = Jout + Kout
    ).
```

Rather than using two variables denoting the input and output of *fibonacci*, state variables offer a more condensed mean of expression. In *fibonacci(!X)*, '!X' is parsed as '!.X' and '!:X', the former being the input and the latter the output.

Example 2.3.3 *Fibonacci written using state variables*

```
:- pred fibonacci(int::in,int::out) is semidet.
fibonacci(!X) :-
    (
        !.X = 0, !:X = 0
    ;
        !.X = 1, !:X = 1
    ;
        !.X > 1,
        J = !.X - 1,
        K = !.X - 2,
        fibonacci(J,Jout),
        fibonacci(K,Kout),
        !:X = Jout + Kout
    ).
```

2.4 Types

The primitive types found in other programming languages are available in Mercury: int, float, string, and char. More elaborated types are available as well: string, array, list, and set. New types called discriminated unions can be constructed [12].

Example 2.4.1 *Discriminated union types*

```
:- type boolean
    ---> true
    ; false.

:- type binaryTree(T)
    ---> leaf(T)
    ; node(binaryTree(T),T,binaryTree(T)).
```

The data constructor is what appears on the right side of the arrow. *boolean* can have two values: *true* or *false*. These are the two data constructors of the type *boolean*. *binaryTree* can be a *leaf* or a *node*. All the nodes and the leaves of the tree are of the same type T.

The fields of a data constructor can be named.

Example 2.4.2 *Discriminated union using named fields in the data constructor*

```
:- type book ---> book(
    book_title :: string,
    book_author :: string,
    book_isbn :: int,
    book_publisher :: string,
    book_year :: int).
```

There are two ways of accessing the fields. The first one is used when more than one field has to be accessed as in the following example [12]:

Example 2.4.3 *Deconstruction of a discriminated union*

```
book(Title,Author,Isbn,Publisher,Year),
if Title = "Parallel Mercury",Author = "Jerome Tannier"
then ...
else ...
```

When only one field has to be accessed, one would do it that way [12]:

```
if book ^ book_title = "Parallel Mercury"
then ...
else ...
```

2.5 Modes

An *inst* is the instantiation state of a variable [12]. The two standard insts available are *free* (uninstantiated) and *ground* (instantiated). From these more elaborate insts can be constructed.

```
:- inst non_empty_list == bound([ground | ground]).
```

This declaration defines an inst which is limited to a list whose head is ground and tail as well. It can also be written in the following way:

```
:- inst non_empty_list ---> [ground | ground].
```

The *mode* of a predicate, or function, is a mapping from the initial state of instantiation of the arguments of the predicate, or the arguments and result of a function, to their final state of instantiation [13].

The two standard modes are:

```
:- mode in == ground >> ground.
:- mode out == free >> ground.
```

Mercury assumes that the input arguments to a function have mode in and the result has mode out [12].

The mode can be supplied in the *pred* declaration or as a separate declaration. The former is a syntactic sugar of the latter.

Example 2.5.1 *Separate pred and mode declaration*

```
:- pred fib(int,int).
:- mode fib(in,out) is det.
```

Example 2.5.2 *pred-mode declaration*

```
:- pred fib(int::in,int::out) is det.
```

The *pred-mode* declaration is only available to predicates with one and one mode only. It is indeed possible to have several modes for a predicate. Each mode corresponds to a particular usage and to a procedure in the compiled code. If the predicate has more than one mode then the use of separate mode declarations is compulsory.

Example 2.5.3 *append predicate with several modes*

```
:- pred append(list(T),list(T),list(T)).
:- mode append(in,in,out) is det.
:- mode append(out,out,in) is multi.
```

append computes the concatenation of two lists. Two cases are considered:

- The user has the two lists from which the concatenation is computed.
- The user has the result of the concatenation and wishes to know what the two lists from which the computation was done are.

The former and the latter correspond respectively to the first and second mode declaration.

2.6 Unique modes

It is sometimes necessary to express the uniqueness of argument variables. To do so, Mercury offers *unique* and *dead* insts. *unique* means that there can only be one reference to the value. *dead* means that there are no references to the corresponding value. The three standard modes using these particular insts are [13]:

```
:- mode uo == free >> unique. % unique output
:- mode ui == unique >> unique. % unique input
:- mode di == unique >> dead. % destructive input
```

Mode *uo* is used to create a unique value. Mode *ui* is used to inspect a unique value without losing its uniqueness. Mode *di* is used to deallocate or reuse the memory occupied by a value that will not be used [13].

However, as we will see in Section 2.7, a predicate might succeed more than once. When the unification of an atom fails, the compiler backtracks to the parent node in the SLD-tree and tries to unify that node with the next clause available. Therefore, the previous clause which the compiler tried unsuccessfully to unify the node with needs to be retained. As a result, *unique* and *dead* insts are not suitable for the task. To that purpose, Mercury provides *mostly_unique* and *mostly_dead* modes which allow the values to be restored [13].

```
:- mode muo == free >> mostly_unique. % mostly unique output
:- mode mui == mostly_unique >> mostly_unique. % mostly unique input
:- mode mdi == mostly_unique >> mostly_dead. % mostly destructive input
```

2.7 Determinism

For each predicate, the user specifies its determinism. It corresponds to the number of times the predicate can succeed. The determinism of a procedure is inferred from the determinism of its goals [15]. The determinism is specified either in the pred-mode declaration or in the mode declaration in case of a separate pred and mode declarations. The following are the possible determinisms [13]:

	Maximum number of solutions		
Can fail?	0	1	>1
no	erroneous	det	multi
yes	failure	semidet	nondet

Table 2.1: Mercury's determinism

Mercury assumes a function as a whole is det [12]. Moreover, *failure* and *erroneous* are two special cases of determinism and are rarely used.

The following is a semidet predicate. It can either fail or succeed exactly once.

Example 2.7.1 *semidet predicate*

```
:- pred string_figure_to_int(string::in,int::out) is semidet.
string_figure_to_int("one",1).
string_figure_to_int("two",2).
string_figure_to_int("three",3).
string_figure_to_int("four",4).
string_figure_to_int("six",6).
string_figure_to_int("seven",7).
string_figure_to_int("eight",8).
string_figure_to_int("nine",9).
```

If the user enters $string\_figure\_to\_int("five", X)$ then it fails. It can not unify with any of the facts following the predicate declaration. $string\_figure\_to\_int("one", X)$ would succeed though and X would unify with 1.

Let's examine Example 2.5.3. According to the second mode declaration, *append* succeeds at least once.

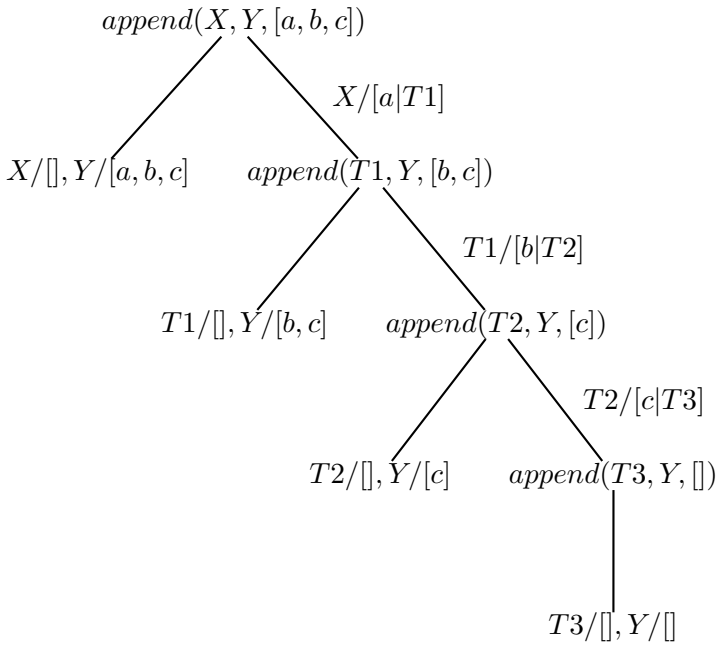


Figure 2.2: SLD tree from Example 2.5.3

With $[a, b, c]$ as the resulting concatenation, several input lists can be computed:

- X = [], Y = [a,b,c]
- X = [a], Y = [b,c]
- X = [a,b], Y = [c]
- X = [a,b,c], Y = []

2.8 Compiler

The Mercury compiler has several back-ends. Some are production ready i.e. low and high-level C. Some are in alpha or beta quality i.e. .NET and Assembler. Others are under development i.e. Java and Mercury byte-code. The most commonly used back-end is the low-level C back-end. Besides being fast, C generated code is also portable and can run on almost all software and hardware platforms.

In order to compile a program, one would invoke 'mmc −−make <*module name*>'.

Example 2.8.1 *Fibonacci compiled and executed:*

```
$ mmc --make fib
$ ./fib 30
832040
```

Programs are compiled using a grade which is a set of options meant for a particular use. These options are related to the target language, the garbage collection strategy, the profiling technique, debugging, the parallel generation of code, ... The entire program must be compiled with the same setting of these options, and it must be linked to a version of the Mercury library which has been compiled with the same setting [16]. This is especially important for large programs written in Mercury like the Mercury compiler itself.

Example 2.8.2 *Fibonacci compiled with the debugging grade*

```
$ mmc --debug fib
```

When the compiler processes a program, it goes through a set of passes. Once all of them are executed, the program is compiled. Mercury's type, mode, and determinism systems are three of these passes and ensure that many of the most trivial programming errors are caught at compile-time rather than at run-time [15]. This speeds up the development of applications. Another pass worth mentioning is the superhomogeneous transformation. It is aimed at transforming the clauses of a Mercury program into a normal form aka superhomogeneous form. In that form, arguments of predicate calls and constructors must be distinct variables, so that unifications implied by multiple occurrences of a variable in a term are made explicit. This greatly simplifies the analysis passes of the compiler [16].

Example 2.8.3 *append before the superhomogeneous transformation*

```
append([],Xs,Xs).
append([X|Xs],B,[X|Zs]) :- append(Xs,B,Zs).
```

Example 2.8.4 *append after the superhomogeneous transformation*

```
append(A,B,C) :- A = [],B = C.
append(A,B,C) :- A = [X|Xs],C = [X|Zs], append(Xs,B,Zs).
```

Mercury's execution mechanism is as follows. Mercury uses eager evaluation aka strict evaluation meaning that an expression is evaluated as soon as it gets bound to a variable [17]. It is in contrast with lazy evaluation aka non strict evaluation meaning that a computation is delayed until such time as the result of the computation is known to be needed [17, 18].

Mercury's atom selection strategy differs whether it is seen from the user or the programmer point of view. When the Mercury compiler processes a program, it tries to reorder the atoms so that the well-modedness rule is respected. This rule goes as follows. For sequential conjunction, it is required that there is an ordering of that conjunction such that every consuming occurrence of a variable comes to the right of a producing occurrence of that variable [15]. Therefore, the user sees Mercury's execution mechanism as a SLD resolution with random atom selection. From a programmer point of view, once the reordering is done, the atom selection is a leftmost process. If the Mercury compiler can not reorder the atoms to match the rule then the program which we try to compile is rejected.

Chapter 3

Previous work on Mercury related to parallelism

Parallelism and concurrency are two notions which are often mixed up. Before we present how parallelism has been implemented in Mercury, it is important to distinguish between the two. Parallelism is a mechanism executing different parts of a program simultaneously on multi-core, multi-processor or distributed systems to enhance the throughput of a program [19]. On the other hand, a concurrent program deals with multiple things happening simultaneously and is not aimed specifically at performance. For instance, a server program can handle multiple clients at the same time [15]. However, concurrency and parallelism are overlapped. Indeed, a program can be concurrent and run in parallel [19].

With the generalization of symmetric multiprocessing (SMP) machines, parallelism has become a major topic of interest in recent years. Parallelism in imperative languages is not an easy thing to do. It is very error-prone due to the presence of side-effects. As defined in [20], *"a side-effect is a change in the state of the program that occurs when evaluating an expression"*. The modification in a function of a global variable or one of its arguments and I/O operations are examples of side-effects [21]. Therefore, effects of parallelism in the imperative paradigm can sometimes be surprising.

Declarative programming languages are known to be better suited to parallel programming involving the decomposition of a task into subtasks and their simultaneous execution on a multi-CPU/core architecture [22]. Indeed, in declarative programming languages, the user describes the "what" a program does and not the "how" it does the computation [15]. Thus, a declarative program can be parallelized without changing its semantics. That coupled with the absence of side-effects make parallelizing a declarative program pretty straightforward contrary to programs written in imperative languages.

In a logic programming setting, it is possible to exploit parallelism in two different ways (some systems exploit both):

- OR-parallelism: If several rules can be used to reduce a goal, then OR-parallelism evaluates these alternatives in parallel rather than waiting for an alternative to fail before trying the next one. OR-parallelism is a direct result of nondeterminism [15].

- AND-parallelism: Goals in a conjunction are reduced in parallel to obtain a speed-up [15].

In the context of Mercury, work on parallelism is limited to AND-parallelism since most Mercury programs make very little use of non-determinism and therefore backtracking. Moreover, backtracking and dependent AND-parallelism (see Section 3.2) do not mix well [15].

Parallelism can be introduced in two different ways [23]:

- Explicitly: the user has to introduce parallelism through the use of an explicit operator. It is up to the user to decide where parallelism would result in a performance gain. The user can thus tweak a program to achieve the maximum performance through the use of an explicit parallel operator.
- Implicitly: parallelism is introduced by the system itself. It requires mechanisms to highlight where parallelism is worth doing. Therefore, the burden of parallelism is shifted from the user to the system. However, that approach might not discover all the parallelism available.

Mercury is particularly well suited for parallelism. Contrary to some other languages in the declarative paradigm, Mercury is a pure declarative language. As said in Section 2.1, even I/O operations are declarative. In other words, programs written in Mercury are free of side effects. Moreover, Mercury's mode system allows the shared variables between parallel goals to be detected at compile-time and, therefore, making the parallelism implementation relatively easy. So far, parallelism in Mercury has been focused on an explicit approach by the introduction of a parallel explicit operator.

3.1 Independent parallelism

The first work on parallelism in Mercury was introduced by Thomas Conway. He modified the Mercury system to allow independent AND-parallelism - the parallel execution of goals which do not have variables in common aka shared variables. The Mercury system is made of three elements: the compiler, the library and the runtime environment. What he did was to split the execution environment into two parts: the engine and the task. The engine supports the evaluation of goals in general. The task is related to the evaluation of a specific goal [11].

The implementation of parallelism has been done in three layers [11]:

- The language layer: Thomas Conway has introduced independent AND-parallelism with an explicit parallel conjunction operator "&". The well modedness rule described in Section 2.8 had to be extended. As stated in [11], *"a parallel conjunction "A & B" is well-moded if the goals "A" and "B" each bind a distinct set of variables, and "B" does not depend on any bindings made in "A" and "A" does not depend on any bindings made in "B" "*. Moreover, Conway has imposed the additional requirement that both goals are deterministic which is assured at compile-time due to the strong determinism system.

  Example 3.1.1 *A non-well-moded parallel conjunction*

```
:- pred main(io::di,io::uo) is det.
    ...
    (
```

```
        foo(A,B)
    &
        foo(B,A)
    )
    ...
:- pred foo(T::in,T::out) is det.
foo(X,Y) :-
    ...
```

That parallel conjunction is not well-moded since the first conjunct depends on the binding made in the second parallel conjunction for the variable A and the second conjunct depends on the binding made in the first conjunct for the variable B. However, the following example contains a parallel conjunction which is well-moded.

Example 3.1.2 *Fibonacci written in Mercury using the parallel operator*

```
:- pred fibonacci(int::in,int::out) is det.

fibonacci(X,Y) :-
    ( X = 0 ->
        Y = 0
    ;
        ( X = 1 ->
            Y = 1
        ;
            ( X > 1 ->
                J = X - 1,
                K = X - 2,
                (
                    fibonacci(J,Jout)
                &
                    fibonacci(K,Kout)
                ),
                Y = Jout + Kout
            ;
                error("fibonacci: wrong value")
            )
        )
    ).
```

In Example 3.1.2, the goals *fibonacci(J,Jout)* and *fibonacci(K,Kout)* are evaluated in parallel. The well-moded rule is respected since *fibonacci(J,Jout)* and *fibonacci(K,Kout)* each bind a distinct set of variables, and *fibonacci(K,Kout)* does not depend on any bindings made in *fibonacci(J,Jout)* and conversely.

- The execution model layer: each processor runs a Mercury engine which picks tasks in a pool of available tasks common to all the Mercury engines. Tasks are executed

independently. Indeed, as described in [11], *"the behaviour of one task does not directly affect the behaviour of another, unless they explicitly communicate"*. The shared address space described in the next layer is used to pass data between the engines and tasks. In Conway's modifications, there is no need for locking data as every variable has a single producer designated at compile-time thanks to the mode system.

- The OS layer: the operating system provides techniques to execute tasks in parallel. On Unix-like platforms, two alternatives are available: processes and threads. As threads are cheaper to create than processes, they are the chosen mechanism for accessing machine parallelism. As stated in [11], *"threads are created within a process, and share the same address space as all the other threads in that process"*. Concurrent access needs synchronization mechanisms which are implemented with mutexes and condition variables.

3.2 Dependent parallelism

Thomas Conway's work was further improved by Peter Wang. He has extended the explicit parallelism approach by making dependent parallelism possible - the parallel execution of goals which have shared variables. As defined in [15], *"shared variables are those which are bound within one parallel conjunct and used in one or more additional parallel conjuncts"*. Every shared variable must be instantiated by exactly one parallel conjunct, the producer, and all other parallel conjuncts which are dependent to the producer must be the consumers of that variable. If a consumer of a shared variable is executed before the variable has been instantiated by a producer, then the execution of the consumer must be suspended until the variable's value has been produced. To minimize those synchronization mechanisms, the parallel conjuncts are reordered such that the producer is the leftmost goal in the parallel conjunction and the consumers are on the right. As in Conway's independent parallelism implementation, goals in a parallel conjunction are restricted to be deterministic [15].

Example 3.2.1 *Dependent parallelism*

```
:- module foo.
:- interface.
:- import_module list.
:- pred foo(list(T)::in,list(T)::in,list(T)::out) is det.

:- implementation
foo(A,B,E) :-
    (
        append(A,B,C)
    &
        append(B,C,D)
    ),
    append(C,D,E).
```

In Example 3.2.1, the variable C is shared between the goals executed in parallel. *append(A,B,C)* is the producer of C and *append(B,C,D)* is the consumer.

Mercury's mode system has been very helpful regarding the implementation of dependent parallelism. Indeed, it provides complete information about producer/consumer relationships so no complicated schemes were needed to determine that information at run-time. The implementation of dependent AND-parallelism goes as follows. A mechanism is needed for a consumer of a shared variable to know if the shared variable has been produced yet and what value it was bound to. To that purpose, Peter Wang has introduced futures objects into the run-time system, which serve as an intermediary - a proxy - between the consumer of a shared variable and the value bound to that variable. Future objects record whether the variable proxied by a future has been bound yet, and if so what value it has been bound to as well as the list of the consumers which are suspended, waiting for the variable to be bound [15].

Four operations are introduced and are defined in [15] as follows:

- *new_future(Future)* creates a future for a shared variable.
- *wait(Future, Value)* causes the caller to suspend if the future is unresolved. Once the future is resolved, execution resumes and the variable *Value* is bound to the value that the future was signalled with.
- *signal(Future, Value)* says that the variable represented by the future has now been instantiated with the value *Value*. Any call to *wait* which was suspended on the future is resumed.
- *lookup(Future, Value)* is equivalent to *wait(Future, Value)* but assumes the future is resolved.

Peter Wang's implementation of dependent AND-parallelism performs a source-to-source transformation of the parallel Mercury program that replaces references to shared variables with references to futures. Instead of accessing to a variable directly, the consumer must ask for its value from a future by calling the built-in operation *wait*. The execution of the consumer is suspended until it knows that the shared variable is instantiated by the producer. Once the variable is bound, the producer calls the built-in *signal* operation on the future associated to the variable. More precisely, in the producer conjunct of a shared variable, a *signal* call is inserted right after the shared variable is instantiated. In conjuncts that consume the shared variable, a *wait* call is inserted right before the first use of the variable in those conjuncts. By signalling as soon as possible and delaying *wait* calls as late as possible, the chance that a future is still unresolved when it is actually needed is minimized so that the maximum amount of parallelism can be extracted out of the code as given [15]. However, these synchronization mechanisms can be costy in time and there might be cases where the sequential execution is actually faster than the dependent parallel execution.

Let's apply the transformation to the code of Example 3.2.1 assuming the following implementation of append:

```
:- pred append(list(T)::in,list(T)::in,list(T)::out) is det.
append(A,B,C) :-
    (
        A = [],
```

```
        B = C
    ;
        A = [X|Xs],
        append(Xs,B,C1),
        C = [X|C1]
    ).
```

Procedure *foo* after inserting the dependent parallelism operations:

```
foo(A,B,E) :-
    new_future(FutureC),
    (
        append(A,B,C),
        signal(FutureC,C)
    &
        wait(FutureC,C),
        append(B,C,D)
    ),
    append(C,D,E).
```

signal(FutureC,C) is added after the first call to *append* as C is a shared variable, produced by *append(A,B,C)*, and consumed by the second call to *append*. *wait(FutureC,C)* is added before the second call to *append* as *append(B,C,D)* needs the value of C to be available before being executed.

Parallel conjuncts may contain procedure calls and these call goals may contain shared variables as procedure arguments. In order to maximize the amount of parallelism we can exploit, *"signal and wait calls need to happen not just before and after the procedure calls that take shared variables as arguments, but within those procedures"* as stated by Peter Wang in [15]. More precisely, the *signal* and *wait* calls must take place respectively right before the producing goal and right after the consuming goal [15].

For that purpose, two specialized version of *append* are created in Example 3.2.1, one whose last argument is a future variable, and the other one whose second last argument is a future variable. The calls to *append* in 3.2.1 are replaced by calls to the specialized versions of *append*. Here is the module *foo* after inserting the dependent parallelism operations within the procedure calls:

```
foo(A,B,E) :-
    new_future(FutureC),
    (
        'Parallel__append__[3]'(A,B,FutureC)
    &
        'Parallel__append[2]'(B,FutureC,D)
    ),
```

```
    append(C,D,E).

:- pred 'Parallel__append[3]'(list(T)::in,list(T)::in,
    future(list(T))::out) is det.
'Parallel__append[3]'(A,B,FutureC) :-
    new_future(FutureC),
    (
        A = [],
        B = C,
        signal(FutureC,C)
    ;
        A = [X|Xs],
        append(Xs,B,C1),
        C = [X|C1],
        signal(FutureC,C)
    ).

:- pred 'Parallel__append[2]'(list(T)::in,future(list(T))::in,
    list(T)::out) is det.
'Parallel__append[2]'(B,FutureC,D) :-
    (
        B = [],
        wait(FutureC,C),
        C = D,
    ;
        B = [X|Xs],
        wait(FutureC,C),
        append(Xs,C,D1),
        D = [X|D1],
    ).
```

There are two specialized versions of append since the future variable is the third argument variable in the first parallel conjunct and the first argument variable in the second parallel conjunct in *foo*. *signal(FutureC,C)* is pushed as early as possible in the call graph i.e. right after C has been produced. In this example, there are no real benefits in moving the *signal* calls in the call graph as *signal(FutureC,C)* is the last goal executed in the specialized version of *append*. However, the wait *calls* are worth moving in a specialized version of *append*. *wait(FutureC,C)* is moved inside *'Parallel__append[2]'* as late as possible i.e. right before C is needed.

'Parallel__append[2]' can be further optimized by using a recursive call instead of *append(Xs,C,D1)*:

```
:- pred 'Parallel__append[2]'(list(T)::in,future(list(T))::in,
    list(T)::out) is det.
'Parallel__append[2]'(B,FutureC,D) :-
```

```
(
    B = [],
    wait(FutureC,C),
    C = D,
;
    B = [X|Xs],
    'Parallel__append[2]'(Xs,FutureC,D1),
    D = [X|D1],
).
```

wait(FutureC,C) is therefore executed at the very latest in the call graph.

In his thesis, Peter Wang benchmarked the current performance of parallelism at that time and indicated that work needed to be done in order to achieve greater performance for parallel execution. Indeed, an excessive amount of parallelism can result in a program running faster sequentially than in parallel because of the overheads associated to the generation of parallel tasks. Therefore, Peter Wang manually modified his test programs so that they would not generate an excessive amount of parallelism and achieve good performance [15]. What we have decided to do is to automate these program transformations. This control of excessive parallelism is called granularity control (see Section 5).

3.3 Deep profiling

Another work worth mentioning in the context of parallelism is the deep profiler. Indeed, as we will see in Section 4, implicit parallelism in Mercury has been implemented through the use of the information generated by such tool. A profiler is a program that examines an application as it runs. The output of a profiler - a profile - contains various information regarding the execution of the profiled program. A profiler is aimed at determining which parts of a program could be optimized. Worrying about performance during coding is usually not a good idea for two main reasons. Firstly, code that has been hand-optimized is harder to debug. It is better to debug and then optimize a program. Secondly, it is hard to guess where the spots worth optimizing in a program are [24].

Profilers used for imperative programs are not adapted to Mercury. The heavy use of recursion, of various forms of polymorphism, and of higher order are what declarative programs differ from their imperative counterparts in. Therefore, a new profiler called deep profiler adapted to the presence of these constructs has been developed.

mdprof, Mercury's deep profiler, implements a source-to-source transformation. When a Mercury program, compiled with the deep profiling grade, is executed, it generates a 'Deep.data' file containing various measurements regarding the execution of the program such as [24]:

- How much time is spent on each part of the program: time taken by the body of the procedure, and time taken by the call tree of the procedure (self and descendants).
- Memory related statistics: number of memory allocations, and amount of memory allocated.
- The number of times each part of the program is executed

Each measurement is taken in a particular context called a deep context, which is the entire list of ancestors of the current predicate/function. That list is made of two nodes, call sites and procedures nodes, which are necessary since call sites do not call other call sites but procedures. Therefore, call site nodes link to procedure nodes and vice versa. In addition, we need to store apart static and dynamic information in order to avoid redundant information. Indeed, it would be wasteful to store for every call site node the line number of the corresponding same call goal in the program source. Therefore, the call graph of a profiled program representing the calling relationships between the subcalls is made out of four data structures which are defined in [24] as follows:

- CallSiteStatic (CSS): CallSiteStatic structures are created by the compiler. There is one CallSiteStatic structure for each call site in the source code. It contains a pointer to the ProcStatic structure of the procedure called at this call site.
- ProcStatic (PS): ProcStatic structures are created by the compiler. There is one ProcStatic structure for each procedure in the source code. It contains an array of CallSiteStatic structures of the call sites within the procedure.
- CallSiteDynamic (CSD): CallSiteDynamic structures are created by the instrumented program during a profiling run. There will be one or more CallSiteDynamic structures for each call site through which the program actually performs a call during the profiling run. For a given call site, there will be distinct CallSiteDynamic structures for each distinct context in which those invocations take place. It contains a pointer to the ProcDynamic structure of the called procedure.
- ProcDynamic (PD): ProcDynamic structures are created by the instrumented program during a profiling run. There will be one or more ProcDynamic structures for each procedure which is called during the profiling run. For a given procedure, there will be distinct ProcDynamic structures for each distinct context in which those calls take place. It contains a pointer to the ProcStatic structure of the procedure that this ProcDynamic structure represents an invocation of. It also contains an array of CallSiteDynamic structures for each call site in the procedure.

Let us consider the Example 3.1.2. Its call graph with the argument parameter 3 is represented on Figure 3.1. At compile-time, three structures are created : one ProcedureStatic and two CallSiteStatic structures. Indeed, in Example 3.1.2, there is only one predicate i.e. *fibonacci* containing two recursive calls. At run time, *fibonacci(3)* is executed as follows. For simplicity reasons, we have decided to ignore the main predicate in which *fibonacci* is called. The *fibonacci(3)* ProcDynamic structure contains pointers to two CallSiteDynamic: one for each of the recursive calls i.e. *fibonacci(2)* and *fibonacci(1)*. These CallSiteDynamic point each to a distinct ProcDynamic strucutre: *fibonacci(2)* and *fibonacci(1)*. Since *fibonacci(1)* is computed without any recursive calls, there is no CallSiteDynamic linked to the *fibonacci(1)* ProcedureDynamic. The same reasoning can be used for the *fibonacci(2)* ProcDynamic structure. For more information on the deep profiler which is not directly relevant to this thesis, the reader may consult [24].

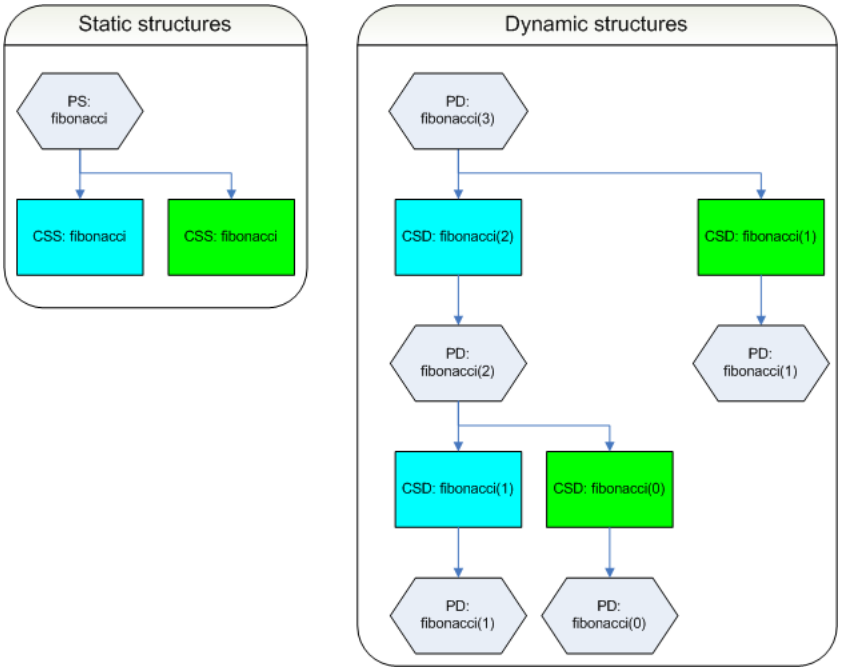


Figure 3.1: Call graph of fibonacci(3)

Chapter 4

Implicit parallelism

During the internship, parallelism in Mercury was further improved. As previously described, parallelism in Mercury consists of using the explicit parallel operator. That mechanism puts the burden of parallelism on the users. What we have decided to implement is an implicit approach. Therefore, it should be up to the compiler to decide where parallelism could be introduced. That approach has been implemented as a two step process. Firstly, a new tool named *mdprof_feedback* highlights the call goals which could be worth parallelizing using the information generated by deep profiling. Secondly, a new pass to the compiler analyses the information generated by *mdprof_feedback* and performs parallel transformations.

4.1 mdprof_feedback

Not every goal in a Mercury program should be parallelized. Indeed, as stated in [25], parallelism incurs overheads, such as those associated with task creation, possible migration of tasks to remote processors, the synchronization mechanisms triggered by shared variables, etc. Therefore, a task for which the costs are larger than the benefits in parallel execution should not be parallelized.

A measure is needed in order to decide whether a call goal would be worthwhile parallelizing. The one we have chosen is the total number of direct and indirect subcalls of a goal call. Indeed, the subcalls of a goal call can in their turn generate subsubcalls and so on. The deep profiler data file generated during the execution of a program compiled in the deep profiling grade contains numerous information. Among others, it contains the number of CallSiteDynamics for a given ProcDynamic. This is not sufficient information. We also need to take into account the number of indirect subcalls. Therefore, a new tool has been developed to extract that information: *mdprof_feedback*. That tool is a new module part of the deep profiler and has to be executed by the user before the program is compiled with the implicit parallelism transformation feature enabled.

As said in Section 3.3, each CallSiteStatic corresponds to a call goal in the Mercury source file. As a CallSiteStatic might have several CallSiteDynamics associated, *mdprof_feedback* computes the average or median number of direct and indirect subcalls aka call sequence count. It only retains the CallSiteStatics for which the average or median call sequence count is above a given threshold. The threshold and the measure type are to be configured by the user respectively with the '−−threshold' and '−−measure' options. The output of *mdprof_feedback* is the profiling feedback file.

Example 4.1.1 *mdprof_feedback executed on the 'Deep.data' file generated by the execution of the deep profiled Fibonacci program with 30 as input*

```
$ mdprof_feedback --threshold 50 --measure average

Profiling feedback file
Version = 1.0
Measure = average
Threshold = 50
fib.main/2-0 0 normal_call fib.fibonacci/2-0
fib.fibonacci/2-0 0 normal_call int.>/2-0
fib.fibonacci/2-0 1 normal_call int.-/3+1-0
fib.fibonacci/2-0 2 normal_call int.-/3+1-0
fib.fibonacci/2-0 3 normal_call fib.fibonacci/2-0
fib.fibonacci/2-0 4 normal_call fib.fibonacci/2-0
fib.fibonacci/2-0 5 normal_call int.+/3+1-0
```

Figure 4.1: Resulting profiling feedback file from Example 4.1.1

The first four lines are the header of the file. The other lines are the CallSiteStatic, sorted according to their occurrence in the Mercury source file, whose average or median call sequence count is above the threshold i.e. 50. The lines are made out of four columns: the caller's name, the slot number, the call type and the callee's name.

The caller is the predicate in which the call is made. The callee is the predicate called. The caller/callee's name is made out of four elements: the module name of the predicate or function to which the procedure belongs, the procedure itself, its arity, and mode number [16].

The slot number is a numerical identifier indicating the position (starting from 0) of the call relative to the other calls in the predicate [16]. Slot numbers are calculated and assigned during the execution of a deep profiled program.

The call type can be one of the following: *normal_call*, *special_call*, *higher_order_call*, *method_call* or *callback*. We will not describe each of them since it is not relevant to this thesis. What we need to know is that *normal_call* is the most common call type.

4.2 Implicit parallelism transformation

The implicit parallelism transformation is a new pass added to the compiler, which decides how to parallelize the CallSiteStatics contained in the profiling feedback file. The implicit parallelism transformation algorithm is presented on Figure 4.2 on page 41. For each CallSiteStatic contained in the profiling feedback file which matches a call goal in a plain conjunction - *FirstGoal*, the implicit parallelism transformation looks in the following goals for another goal which could be worth parallelizing - *LastGoal*. The implicit parallelism transformation can hit one of the following goals:

- A switch or an "if then else" goal: *FirstGoal* can not be parallelized. Indeed, we only want to parallelize calls among themselves since we have no information regarding the sufficient amount of work for parallelism to be worthwhile for switch or "if then else" goals.
- A parallel conjunction: *FirstGoal* might be added to the first conjunct of that conjunction.
- A goal call which is contained in the profiling feedback file: we might create a parallel conjunction with two conjuncts, one for each call goal.
- Another type of goal: the algorithm keeps on searching in the next goals for a goal to be parallelized with *FirstGoal*.

If the implicit parallelism transformation has searched all of the goals following *FirstGoal* and has not found a goal call contained in the profiling feedback file or a parallel conjunction then no parallelism is possible. When the algorithm finds *LastGoal*, there might be other goals between that goal and *FirstGoal*. Since these middle goals are not worth parallelizing and might be introducing more shared variables than there already are between *FirstGoal* and *LastGoal*, we want to exclude them from the parallel conjunction that we are about to create/update. More precisely, our aim is to place the middle goals right before the parallel conjunction happens. To do so, we check whether or not they are dependent to *FirstGoal*. If they are then they can not be moved. Therefore, we can not parallelize *FirstGoal* and *LastGoal*. To illustrate two calls which can not be parallelized, let us consider the following example:

```
    ...
    append(A,B,C),
    C1 = [45|C],
    append(A,C1,D),
    ...
: pred append(list(T)::in,list(T)::in,list(T)::out) is det.
    ...
```

append(A,B,C) and *append(A,C1,D)* are respectively *FirstGoal* and *LastGoal*. *C1 = [45|C]* is dependent to *FirstGoal*. Therefore, the former can not be moved before the latter and, as a result, *FirstGoal* and *LastGoal* can not to be parallelized.

If the goals in the middle - *MiddleGoals* - are dependent to *FirstGoal* then we could have checked the dependency to *LastGoal* and, if they were independent, move *MiddleGoals* right after the parallel conjunction newly created/updated. This would have increased our chances of moving *MiddleGoals* either before the newly created/updated parallel conjunction or after it. In the current implementation of the implicit parallelism transformation, we have limited our dependency checks to *FirstGoal*.

Once we know that there are no goals in the middle or that the goals in the middle are independent to the first goal, we check if *FirstGoal* and *LastGoal* are worth parallelizing. To do so, we check the number of shared variables. As said in Section 3.2, dependent parallelism

implies synchronization mechanisms which slow down the execution. What we want to avoid is a situation where the sequential execution is actually faster than the parallel execution. Therefore, we do as follows.

A call and a parallel conjunction are worth parallelizing if the number of shared variables is strictly smaller by N than the number of argument variables of the call. Indeed, if the number of shared variables was equal to the number of argument variables of the call goal then each of them would require synchronization mechanisms. We make an assumption that it would slow down the execution to a point where there would not be any benefits in parallelizing the goals. N has to be determined so that, on average, the goals are worth parallelizing. In the current implementation of the implicit parallelism transformation implementation, N has been fixed to 1. To illustrate a call and a parallel conjunction which are worth parallelizing, let us consider the following example:

```
:- pred foo(T::in,T::out,T::out) is det.
foo :-
    ...
    A = ...,
    foo(A,B,C),
    (
        fooConsumer(B)
    &
        fooConsumer(C)
    ).

:- pred fooConsumer(T::in) is det.
    ...
```

foo(A,B,C) and the parallel conjunction have two shared variables: *B* and *C*. Therefore, synchronization mechanisms will be used for these two variables. However, the variable *A* does not require that since it is not shared with the parallel conjunction.

Two calls are worth parallelizing if the number of shared variables is strictly smaller by N than the number of argument variables of at least one of the two calls. Indeed, if the number of shared variables was equal to the number of argument variables for each of the two call goals then each variable would require synchronization mechanisms. For the same reasons as above, this would not be a desirable situation. Let us consider the following code fragment:

```
    ...
    append(A,B,C),
    append(A,C,D),
    ...
```

where *append* is defined as follows:

```
: pred append(list(T)::in,list(T)::in,list(T)::out) is det.
```

append(A,B,C) and *append(A,C,D)* are worth parallelizing since the only shared variable is *C*. It is produced by *append(A,B,C)* and consumed by *append(A,C,D)*.

If *FirstGoal* and *LastGoal* are worth parallelizing then we parallelize them as follows. If *LastGoal* is a call then we create a parallel conjunction which contains these two goals after the goals in the middle. If *LastGoal* is a parallel conjunction then we move *FirstGoal* into the first conjunct of the parallel conjunction.

The implicit parallelism algorithm cases are listed on Figure 4.3 on page 42. The cases for which the two calls can and can not be parallelized are indicated respectively with *OK* and *KO*. Let us illustrate those cases with the following predicates:

```
:- pred fooChildProducer(list(T)::out,list(T)::out) is det
fooChildProducer(A,B) :-
    ...
:- pred fooChildConsumer(list(T)::in,list(T)::in) is det
fooChildConsumer(A,B) :-
    ...
```

We assume that all calls to *fooChildProducer* and *fooChildConsumer* predicates are in the profiling feedback file.

Example 4.2.1 *Case 1: No middle goals, not worth parallelizing*

```
:- pred foo() is det.
foo() :-
    fooChildProducer(A,B),
    fooChildConsumer(A,B),
    ...
```

fooChildProducer(A,B) and *fooChildConsumer(A,B)* are not worth parallelizing since there are two shared variables and two argument variables for each of the call.

Example 4.2.2 *Case 2: No middle goals, worth parallelizing, last goal is a parallel conjunction*

```
% Before the implicit parallelism transformation
:- pred foo(list(T)::in,list(T)::in) is det.
foo(A,B) :-
    fooChildProducer(C,D),
    (
        fooChildProducer(E,F)
    &
        fooChildConsumer(A,B)
    ),
```

```
    ...

% After the implicit parallelism transformation
:- pred foo(list(T)::in,list(T)::in) is det.
foo(A,B) :-
    (
        fooChildProducer(C,D),
        fooChildProducer(E,F)
    &
        fooChildConsumer(A,B)
    ),
    ...
```

fooChildProducer(C,D) and the parallel conjunction are worth parallelizing as there are no shared variables between the two. *fooChildProducer(C,D)* is therefore moved into the first parallel conjunct.

Example 4.2.3 *Case 3: No middle goals, worth parallelizing, last goal is a call*

```
% Before the implicit parallelism transformation
:- pred foo() is det.
foo() :-
    fooChildProducer(A,B),
    fooChildProducer(C,D)
    ...

% After the implicit parallelism transformation
:- pred foo() is det.
foo() :-
    (
        fooChildProducer(A,B)
    &
        fooChildProducer(C,D)
    ),
    ...
```

fooChildProducer(A,B) and *fooChildProducer(C,D)* are worth parallelizing as there are no shared variables between the two.

Example 4.2.4 *Case 4: Middle goals dependent to the first goal*

```
:- pred foo() is det.
foo() :-
    fooChildProducer(A,B),
    append(B, [56], B2),
    fooChildConsumer(A,B2),
    ...
```

The middle goal *append(B, [56], B2)* is dependent to *fooChildProducer(A,B)* since B is a shared variable produced by *fooChildProducer* and consumed by *append*. Therefore, we can not parallelize *fooChildProducer(A,B)* and *fooChildConsumer(A,B2)*.

Example 4.2.5 *Case 5: Middle goals independent to the first goal, not worth parallelizing*

```
:- pred foo(int::in) is det.
foo(I) :-
    fooChildProducer(A,B),
    I = I + 1,
    fooChildConsumer(A,B),
    ...
```

The middle goal $I = I + 1$ is independent to *fooChildProducer(A,B)*. However, *fooChildProducer(A,B)* and *fooChildConsumer(A,B)* are not worth parallelizing since there are two shared variables and two argument variables for each of the call.

Example 4.2.6 *Case 6: Middle goals independent to the first goal, worth parallelizing, last goal is a parallel conjunction*

```
% Before the implicit parallelism transformation
:- pred foo(list(T)::in,list(T)::in, int::in) is det.
foo(A,B, I) :-
    fooChildProducer(C,D),
    I = I + 1,
    (
        fooChildProducer(E,F)
    &
        fooChildConsumer(A,B)
    ),
    ...

% After the implicit parallelism transformation
:- pred foo(list(T)::in,list(T)::in, int::in) is det.
foo(A,B, I) :-
    I = I + 1,
    (
        fooChildProducer(C,D),
        fooChildProducer(E,F)
    &
        fooChildConsumer(A,B)
    ),
    ...
```

The middle goal $I = I + 1$ is independent to *fooChildProducer(C,D)*. *fooChildProducer(C,D)* and the parallel conjunction are worth parallelizing since there are no shared variables. *fooChildProducer(C,D)* is moved inside the first conjunct of the parallel conjunction.

Example 4.2.7 *Case 7: Middle goals independent to the first goal, worth parallelizing, last goal is a call*

```
% Before the implicit parallelism transformation
:- pred foo(int:in) is det.
foo(I) :-
    fooChildProducer(A,B),
    I = I + 1,
    fooChildProducer(C,D)
    ...

% After the implicit parallelism transformation
:- pred foo(int:in) is det.
foo(I) :-
    I = I + 1,
    (
        fooChildProducer(A,B)
    &
        fooChildProducer(C,D)
    ),
    ...
```

The middle goal $I = I + 1$ is independent to *fooChildProducer(A,B)*. *fooChildProducer(A,B)* and *fooChildProducer(C,D)* are worth parallelizing since there are no shared variables. A parallel conjunction is created with the two calls. $I = I + 1$ is placed right before it.

For each plain conjunction containing the goals $Goal_1$, $Goal_2$, ..., $Goal_n$:

- Find a goal call $Goal_i$ which is contained in the profiling feedback file. That goal is named *FirstGoal*.
- In $Goal_{i+1}$, $Goal_{i+2}$, ..., $Goal_n$, we search sequentially for another goal - named *LastGoal* - wich could be parallelized with *FirstGoal*:
  - $Goal_j$ is a goal call which is contained in the profiling feedback file. We might use that goal for introducing parallelism.
  - $Goal_j$ is a parallel conjunction. We might use that goal for introducing more parallelism.
  - $Goal_j$ is a switch or an "if then else" goal. *FirstGoal* can not be parallelized.
  - We have reached the end of the plain conjunction. *FirstGoal* can not be parallelized.
- If *MiddleGoals* - the goals between *FirstGoal* and *LastGoal* - are dependent to *FirstGoal*, then *FirstGoal* and *LastGoal* can not be parallelized.
- We check if *FirstGoal* and *LastGoal* are worth parallelizing. If they are, then we parallelize the two goals.

Figure 4.2: Implicit parallelism transformation algorithm

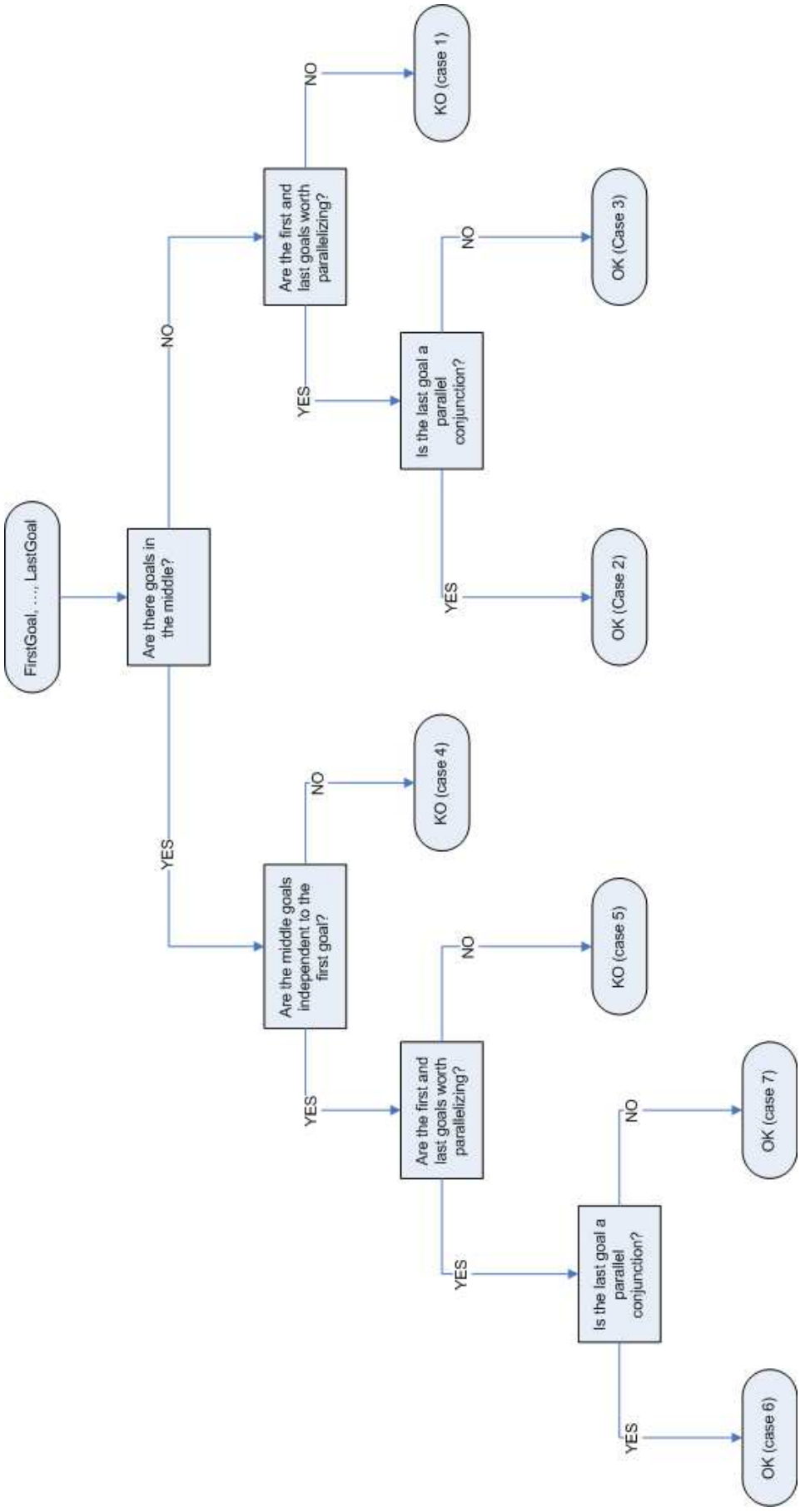


Figure 4.3: Implicit parallelism algorithm cases

Chapter 5

Granularity control

Granularity control aims at limiting the excessive generation of parallelism by executing a task sequentially, which was meant to be parallelized, if the gain obtained by executing the task in parallel is less than the overheads required to support its parallel execution. Indeed, parallelism incurs two kinds of overhead such as described in [26]:

- *Direct parallel task execution overheads* correspond to the cost of creating and maintaining a parallel task. Direct overheads are either fixed or proportional to the amount of work performed by a task - the task size - which is evaluated by the input data size. They do not vary greatly from program to program.

- *Indirect parallel task execution overheads* correspond to the creation of subtasks dynamically. Indirect overheads can contribute significantly to the total overhead. In contrast to direct overheads, indirect overheads may not be very dependent on the size of the task being executed in parallel and, in addition, may vary greatly from programs to programs.

As the amount of overhead is architecture dependent, the benefits of controlling granularity will be higher for systems with greater parallel execution overheads [25]. Granularity control itself can induce new overheads. In order to minimize them, granularity control is usually implemented in the following way. It consists in applying program transformations at compile-time which do not depend on the run-time context and determining at run-time what can only be known during the execution of the program i.e. dynamic information such as the number of free CPUs [26].

Such granularity control can be implemented in various ways. The most popular one is through static analysis. Granularity control through static analysis is based on the task size. It consists in generating at compile-time cost functions which estimate the amount of work of the predicates and functions of programs. Those cost functions are evaluated at run-time when the input data size of the predicates and functions of the original program is know [1]. However, task size does not capture the full complexities of overheads in parallel execution as it is not appropriate for measuring indirect overheads. As stated in [26], we need a metric that could be applied to cases where it is hard to estimate time-complexity, and to cases where subtasks creation is unevenly balanced. Therefore, we have decided to use the distance metric, defined in [26] as *"the amount of work performed between successive points at which major parallel overheads are incurred"*.

5.1 Distance transformation

The granularity control using the distance metric can either happen at compile-time or at run-time. We have decided to implement it at compile-time for the following reasons. Firstly, as said above, we want to do as much of the work at compile-time. Secondly, the distance granularity transformation was easier to implement at compile-time than at run-time. Finally, the distance granularity control at compile-time provides on average the same benefits as the one at run-time according to [26]. Therefore, the distance granularity transformation has been implemented as a new pass added to the compiler.

The distance transformation only applies to recursive predicates or functions. A specialized version of the original recursive predicate which we apply the transformation to is created. We add an extra argument to that specialized version of the predicate. That argument variable controls the frequency at which the recursive calls are executed in parallel - the distance between parallel tasks - and, therefore, allows to limit the amount of overhead, associated to parallelism, being generated in programs. In the original version of the predicate, the recursive calls are changed to call the specialized predicate.

Let us look at the following program to see what the transformation does:

Example 5.1.1 *Distance transformation applied to Fibonacci: original predicate updated*

```
:- pred fibonacci(int::in,int::out) is det.

fibonacci(X,Y) :-
    ( X = 0 ->
        Y = 0
    ;
        ( X = 1 ->
            Y = 1
        ;
            ( X > 1 ->
                J = X - 1,
                K = X - 2,
                (
                    DistanceGranularityFor__pred__fibonacci__2(J,Jout,2)
                &
                    DistanceGranularityFor__pred__fibonacci__2(K,Kout,2)
                ),
                Y = Jout + Kout
            ;
                error("fibonacci: wrong value")
            )
        )
    ).
```

DistanceGranularityFor__pred__fibonacci__2 is the specialized version of *fibonacci*. The last argument is the one which controls the distance between parallel executions. The distance has been set to 2 in this example through the use of '−−distance-granularity $<$ *int value* $>$' option. Now let us have a look at the specialized predicate.

Example 5.1.2 *Distance transformation applied to Fibonacci: specialized predicate*

```
:- pred DistanceGranularityFor__pred__fibonacci__2(int::in,int::out,int::in)
    is det.

DistanceGranularityFor__pred__fibonacci__2(X,Y,Distance) :-
    ( X = 0 ->
        Y = 0
    ;
        ( X = 1 ->
            Y = 1
        ;
            ( X > 1 ->
                J = X - 1,
                K = X - 2,
                ( Distance = 1 ->
                    (
                        DistanceGranularityFor__pred__fibonacci__2(J,Jout,2)
                    &
                        DistanceGranularityFor__pred__fibonacci__2(K,Kout,2)
                    )
                ;
                    DistanceGranularityFor__pred__fibonacci__2(J,Jout,Distance - 1),
                    DistanceGranularityFor__pred__fibonacci__2(K,Kout,Distance - 1)
                ),
                Y = Jout + Kout
            ;
                error("fibonacci: wrong value")
            )
        )
    ).
```

When the fibonacci predicate is executed, the recursive calls to the specialized version are executed in parallel. Once we enter that specialized predicate, the recursive calls are run sequentially until the extra argument to the call equals the value 1. Then, the recursive calls are run in parallel and the granularity variable is restored to its original value i.e. 2 in the example. In other words, AND-parallelism is generated for every two calls to *DistanceGranularityFor__pred__fibonacci__2*. That allows to reduce the number of subtasks and, therefore, the overheads associated with each of them.

Let us consider the execution of *fibonacci(5)* with a distance value of 2. Here is its call graph:

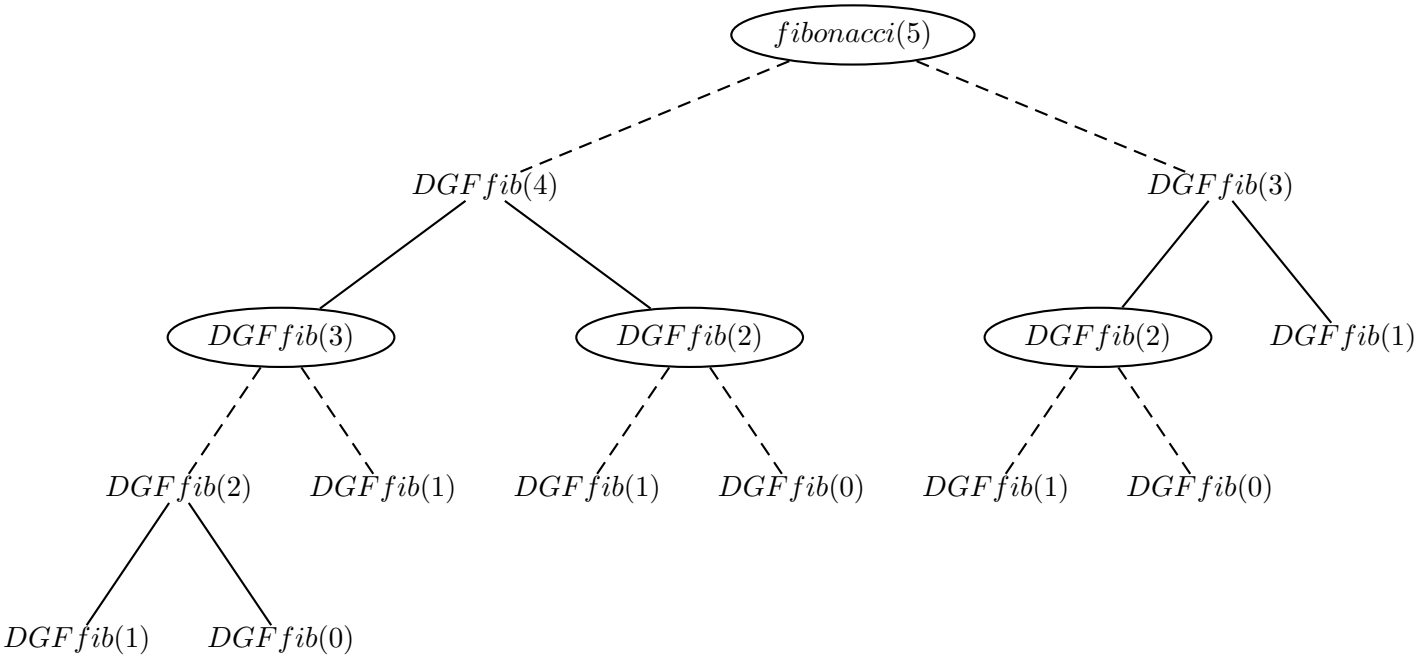


Figure 5.1: Call graph of fibonacci(5)

DGFfib stands for *DistanceGranularityFor__pred__fibonacci__2*. In this example, the recursive calls are executed in parallel every two times since the distance value is 2. The dashed lines represent the parallel branches. These are the recursive calls which are executed in parallel. The straight lines represent the plain conjuncts i.e. the recursive calls executed sequentially.

Since we have implemented our granularity control system at compile-time, it is unable to take into account dynamic information which can only be determined at run-time such as the number of free CPUs. Therefore, a granularity control should also be happening at run-time. It has actually already been partially implemented by Zoltan Somogyi. His system prevents the creation of a higher number of tasks than the number of available CPUs which can only be known at run-time.

5.2 Benchmarks

In order to measure the gains obtained from the distance granularity control transformation, we have run a couple of tests. The machine we have to our disposal is a PC with 64-bit Pentium D 3.0 GHz, dual core CPU with 4 GB RAM. The machine runs on Linux 2.6.12.5.

The test programs we have chosen are fibonacci, quicksort, and hanoi. They have been compiled and executed with the minimum required set of options i.e. with parallel generation of code and, of course, distance granularity control enabled. No other options have been set so that the benefits of using the distance granularity control are those of the average user. The test programs have been run when the machine was free. In the following tables, *np* and *ngc* are respectively the non parallel execution of the test program and its parallel execution with no granularity control. Each test program was run 6 times. We have discarded the best

and worst times and done the average on the remaining four times. The tables are made out of four columns: the distance value, the time taken for the execution of the test program, the speed-up compared to the the non parallel execution, and the speed-up compared to the parallel execution with no granularity control.

distance	Time (s)	Speed-up (np)	Speed-up (ngc)
np	1.75	1.0	1.14
ngc	2.00	0.88	1.0
10	1.96	0.89	1.02
50	1.92	0.91	1.04
100	1.90	0.92	1.05
250	1.90	0.92	1.05
500	1.91	0.92	1.05
1000	1.93	0.91	1.04

Table 5.1: Benchmark results for quicksort(1000000)

quicksort(1000000) generates a list of 1 000 000 random integers and then, applies the quick sort algorithm to that list. The time needed to generate the list is not taken into account. The parallel execution with no granularity control takes 14% more time to compute than the sequential execution. Granularity control allows to improve the performance of the parallel execution. The distance value providing the best results reduces the execution time by 5% compared to the parallel execution with no granularity control. Notice that *quicksort(1000000)* is faster when run sequentially whatever the distance value is.

distance	Time (s)	Speed-up (np)	Speed-up (ngc)
np	6.61	1.0	1.09
ngc	7.20	0.91	1.00
2	6.72	0.98	1.07
3	6.67	0.99	1.08
4	6.48	1.02	1.11
5	6.46	1.02	1.11
10	6.35	1.04	1.13
15	6.47	1.02	1.11
17	6.61	1.0	1.09

Table 5.2: Benchmark results for hanoi(22)

With no granularity control, the parallel execution of *hanoi(22)* takes 9% more time to compute than the sequential execution. This results from the excessive amount of parallelism generated and the overheads incurred. The distance providing the best results for parallel execution is 10. With that setting, the perfomance is of a 1.13 factor compared to the non granularity control execution and the execution time is slightly faster than the non parallel's.

distance	Time (s)	Speed-up (np)	Speed-up (ngc)
np	3.09	1.0	12.70
ngc	39.23	0.07	1.00
2	16.05	0.19	2.44
5	8.27	0.37	4.74
10	3.61	0.86	10.87
15	2.53	1.22	15.51
17	2.30	1.34	17.06
20	5.29	0.58	7.42

Table 5.3: Benchmark results for fibonacci(40)

With granularity control disabled, *fib(40)* takes 12 times more time to compute. Compared to *qsort(1000000)* and *hanoi(22)*, the difference between the sequential execution and the parallel execution with no granularity control is much more important. That is because *fib(40)* do almost no work apart from making recursive calls contrary to the two other tests programs. Therefore, parallelism occurs more often and so do the associated overheads. When granularity control is enabled, the distance providing the best results for *fib(40)* is 17. With that setting, the performance is of a 17.06 factor compared to the non granularity control execution. The parallel execution with a granularity control distance of 17 reduces the execution time by 26% compared to the sequential execution.

The distance granularity control transformation shows important speed-ups for recursive predicates which do little work around the recursive calls. For other predicates, the gains are less impressive. Our current implementation of the distance granularity control needs further improvements. In a future version of the Mercury compiler, it would be interesting to let the compiler decide what the optimal distance value should be. As the the creation of parallel work can occur at irregular intervals, it should be the average distance being considered [26]. This could be computed from the 'Deep.data' file generated from the execution of a deep profiled program. Also, the distance granularity control has the major drawback of being limited to recursive predicates. Therefore, it can not control all of the excessive parallelism being generated. Granularity control through static analysis could be implemented and its performance benefits be compared to the ones of the distance granularity control even though it is claimed in [26] not to be the best approach.

Chapter 6

Parallelism in other programming languages

The declarative paradigm accounts for a large number of languages. Among them, there are Mercury, Haskell and Prolog. In what follows, we will compare the former with the other two mainly on how parallelism has been implemented.

6.1 Haskell

In the late 80's, there existed more than a dozen functional programming languages. At the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon, it was decided that a committee should be formed to define an open standard for such languages. The first version of that standard, named Haskell, was defined in 1990. In late 1997, Hakell 98 was released specifying a stable, minimal, portable version of the language and an accompanying standard library. At present, the most well-known implementations of the standard are Hugs - Haskell User's Gofer System - and GHC - Glasgow Haskell Compiler [27].

Haskell is a lazy programming language. Lazy evaluation has the following advantages over Mercury's eager evaluation strategy such as described in [27]: *"performance increases due to avoiding unnecessary calculations, avoiding error conditions in the evaluation of compound expressions, the ability to construct infinite data structures, and the ability to define control structures as regular functions rather than built-in primitives"*.

However, lazy evaluation also has some significant disadvantages as stated by Julien Fischer, member of the Mercury team. Firstly, lazy evaluation is hard to implement in an efficient way. Indeed, lazy evaluation can cause a substantial amount of overhead in the execution of functional programs especially in case of backtracking. Although modern compilers do a pretty good job with regard to optimizing lazy functions, they do a bad job of optimizing away the unnecessary laziness of lazy data structures. Secondly, laziness makes the operational semantics much more complicated. As a result, debugging and reasoning about performance are much harder [28, 29].

Since Haskell is a pure functional programming language and therefore deterministic [1], only AND-parallelism can be exploited. One of the many parallel Haskell implementations

[1] Non-determinism can actually be introduced using monads, a type constructor. See [30] for details.

is GPH - Glasgow Parallel Haskell. It is an extension of Haskell, adding a primitive for parallel composition *par*, that is used together with sequential composition *seq* to express how a program should be evaluated in parallel. GPH is a semi-implicit approach. Indeed, it is required that the programmer indicates those expressions than can be evaluated in parallel but the run-time environment is free to ignore any available parallelism [31, 32]. Mercury's parallelism approach is similar. It exploits mainly explicit parallelism through the use of a parallel operator. Indeed, as things are, the implicit transformation is not mature yet as it requires the intervention of the user which has to execute the deep profiled version of the program before applying the implicit parallelism transformation to it. However, Mercury differs to GPH with regards to the execution of parallel programs. Indeed, the execution model layer in Mercury does not dismiss any parallel operators which have been specified by the user or by the implicit module transformation.

GUM - Graph reduction for a Unified Machine model - is the name of the run-time system that provides support on parallel architectures [33]. It is equivalent to the execution model layer in Mercury. As stated in [34], *"GUM is message based, and uses PVM, a communication infrastructure available on almost every multi-processor"*. GUM's architecture is based on a collection of processor-memory units (PEs) connected by some kind of network accessible through PVM [34]. A parallel task in GUM is represented by a spark. Once a spark has been turned into a thread, or been activated by a PE, the thread will remain on this PE. The sparks are created by executing the *par* primitive on a CPU and stored in the spark pool. At start-up, a GUM program creates a PVM manager which is responsible for mapping the sparks to the available PEs. Once a PE has no more local sparks available, then it looks for work in others PEs by sending a FISH message to a randomly chosen PE [34]. The PE receiving the message searches for a spark in its local spark pool and, if available, send it to the requesting PE. Otherwise, the FISH message will be forwarded to another PE till it finds a PE with a spark available. As stated in [31], *"that mechanism is usually called work stealing or passive load distribution"*. A parallel program in GPH terminates when the main PE completes or encounters an error. A FINISH message is sent to the main PE, which is then forwarded to the other PEs [34]. A work-stealing approach is also being considered for the Mercury execution layer. Peter Wang has recently modified the run-time layer such that each engine has its own local spark pool. As suggested in [15], an idle engine could steal work from another engine.

In GPH, each closure - *"a function paired with an environment consisting of the variables needed for the evaluation of the function"* as defined in [27] - has an address which can be globalized. In that case, the closure being executed by a thread on a PE is visible to all the other PEs. When a thread enters a closure which is not local to its PE - a FETCH-ME closure, the closure is then evaluated on another PE. Therefore, the former needs to know what the result of the foreign closure is before continuing its execution. The execution of the thread is suspended. A FETCH message is sent to the PE which owns the closure meaning that a thread on that PE is in charge of the evaluation of the closure. When the closure is evaluated, a RESUME message is sent to the suspended thread with all the necessary information to continue its execution [34].

This is how dependent AND-parallelism is performed in Haskell. When compared to Mercury, there is no need for a mode system to determine the producer/consumer relationships since Haskell is a functional programming language and therefore, the input and output arguments of a function can be determined straightforwardly at compile-time. However, Mercury is more than a functional programming language. It is also a logic programming

language. The association of the two paradigms makes the mode system necessary in order to determine at compile-time the producer/consumer relationships rather than at run-time like in Prolog.

6.2 Prolog

Prolog which stands for PROgramming in LOGic was created around 1972 by Philippe Roussel and Alain Colmerauer. Robert Kowalski contributed to the theoretical framework on which Prolog is founded i.e. the procedural interpretation of Horn clauses. Roussels and Colmerauer's objective was to consider first order logic as a declarative programming language but also as a knowledge representation language [27, 35, 36].

Today's major implementations of Prolog are GNU Prolog, Quintus Prolog, SICStus Prolog, Strawberry Prolog, SWI-Prolog, YAP Prolog, and Cia Prolog [27, 37]. In this thesis, we will look into a parallel implementation of Prolog: ACE - And/Or-parallel Copying-based Execution of Logic Programs. It exploits all forms of parallelism i.e. independent AND-parallelism, dependent AND-parallelism, and OR-parallelism in an implicit way. The ACE system is extremely efficient and has shown excellent speed-ups on a variety of benchmarks [38].

Independent AND-OR parallelism

Due to the impurity of the language and the lack of mode information, independent AND-parallelism in Prolog can not be detected at compile-time. Therefore, ACE generates Conditional Graph Expressions (CGEs) which are described by Peter Wang in [15] as *"simple tests created by the compiler and evaluated at run-time to decide if a goal should be executed in parallel or not"*. They are of the following form:

$$(condition \Rightarrow goal_1 \ \& \ goal_2 \ \& \ ... \ \& \ goal_n)$$

stating that if *condition* is true then the goals $goal_1$, $goal_2$, ..., $goal_n$ are to be evaluated in parallel, otherwise they are to be evaluated sequentially [1].

Contrary to Mercury, ACE also supports OR-parallelism which is a direct consequence of non-determinism. In Mercury, since parallel goals are restricted to be deterministic, OR-parallelism can not occur. Allowing non-deterministic goals to be run in parallel expands the amount of parallelism which can be exploited but severely increases the complexity of the parallel implementation as we will see.

ACE's execution mechanism is expressed through Extended AND-OR Trees and Composition Trees. An Extended AND-OR tree is a traditional AND-OR tree with solution sharing. Consider the CGE ($true \Rightarrow b \ \& \ c$), where b and c have respectively m and n OR-parallel solutions each, and *true* meaning that b and c can be evaluated in parallel unconditionally. Figure 6.1 is the resulting Extended AND-OR Tree from the query '$? - a, (b \ \& \ c), d$' [1].

Instead of computing the solutions of goal c for every solution found in goal b, the solutions for b and c are computed only once since b and c are independent. Once the distinct solutions are found, all of the possible pair of solutions for goal b and c i.e. the CGE are constructed. That operation is called the cross-product.

However, solution sharing can not be used if goals have side-effects or extra-logical calls in them. Indeed, if goal c of the above query was impure, then its recomputation might not

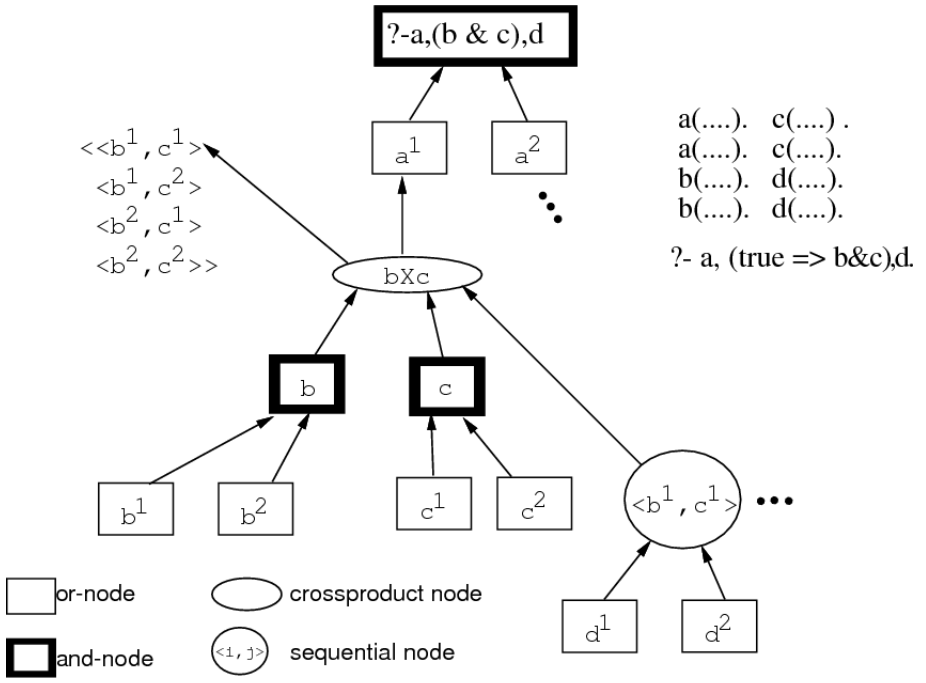


Figure 6.1: Extended AND-OR Tree from [1]

produce the same set of solutions. Therefore, for every alternative of goal b, goal c needs to be recomputed. If we did not do so, then every alternative of goal c would have the same binding produced by goal b associated. Goal recomputation ensures that the parallel execution shows the same external behaviour as the one of the sequential execution [1].

Goal recomputation is represented using Composition Trees aka C-tree. Figure 6.2 is the C-tree of the CGE ($true \Rightarrow a \ \& \ b$) where a and b are impure. For each alternative of AND-parallel goal a, goal b is computed in its entirely [1].

Dependent AND-OR parallelism

ACE parallelism implementation has been further improved by introducing dependent AND-parallelism. It differs to Mercury's dependent AND-parallelism implementation in the identification of the producer/consumer relationships. Indeed, they can only be determined at run-time due to the absence of a mode system in ACE. Furthermore, non-determinism in ACE increases the complexity of a successful implementation of dependent AND-parallelism.

Consider the goals *p(X)*, *q(X)* with the following definition:

```
p(X) :- g1, X = 1, s1.
p(X) :- g2, X = 2, s2.
p(X) :- g3, X = 3, s3.

q(3) :- h1.
q(4) :- h2.
q(5) :- h3.
```

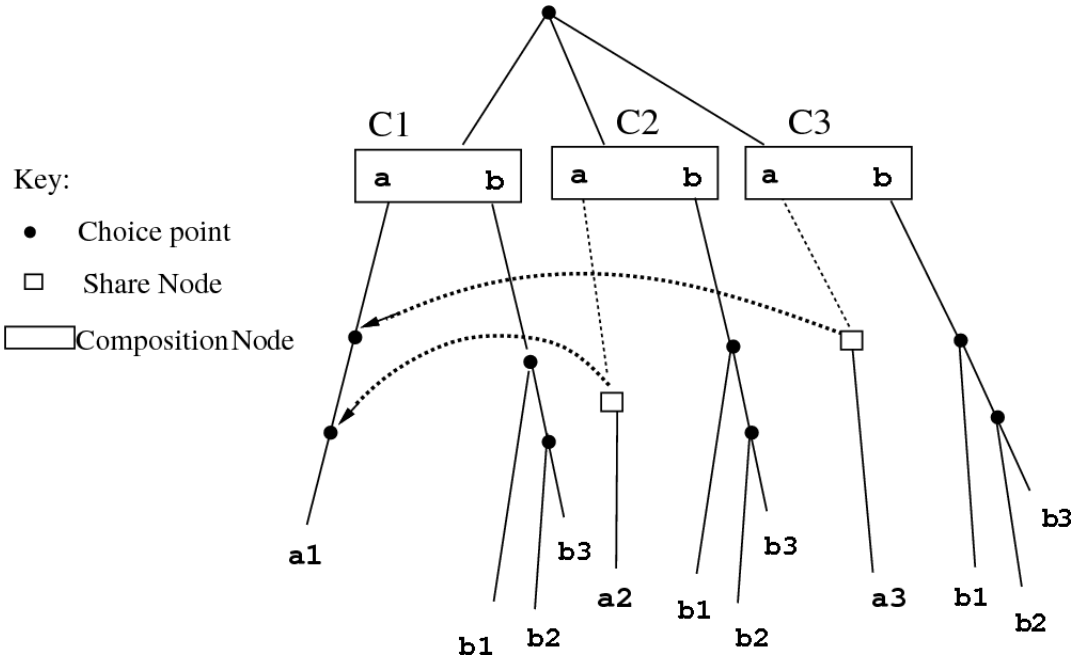


Figure 6.2: Composition Tree from [1]

p and q are non-deterministic. If we execute *p(X)* and *q(X)* sequentially, then only *h1* will be executed. Indeed, the only solution possible for X is 3. However, if we execute those two goals in parallel, then each of them would produce 3 bindings. Compared to sequential execution, dependent AND-parallel execution performed redundant execution of *h2* and *h3*. If we execute q only after a binding for X is produced by p then parallelism narrows down to a sequential execution. Therefore, we need to strike a balance between the two [39].

There are various ways of dealing with the problem of producer/consumer relationships. The first one is DDAP (Dynamic Dependent And-Parallelism). It consists in an approximation of the identification of the producer. At the beginning of the execution of parallel goals, the leftmost goal that can access a shared variable X is considered to be the producer. The other goals on the right are suspended till the leftmost goal produces the variable X. If it turns out that that goal terminates without instantiating X then the next leftmost goal than can access X is designated as the producer and its execution is resumed. This mechanism may thus results in a loss of parallelism which could be exploited [39, 40].

Dependent AND-parallelism in ACE has been implemented using a EDDAP (Extended Dynamic Dependent And-Parallelism) mechanism. It is an improvement over the DDAP mechanism. The execution of EDDAP goes as follows. A goal designated as a consumer can bind a shared variable X if the goal is deterministic. Consider the following example [39]:

```
p(X) :- g1, X = 1, s1.
p(X) :- g2, X = 2, s2.
p(X) :- g3, X = 3, s3.

q(X) :- <deterministic computation>, X = 2.
```

When *p(X)* and *q(X)* are executed in parallel using DDAP, the leftmost goal p is considered to be the producer unless it finishes without instantiating the shared variable X and q is the consumer which is suspended until p produces the binding. Using that mechanism, the bindings of X with 1 and 3 would fail since 2 is the only binding possible. In EDDAP, the execution of the consumer goal q will not be suspended since it is deterministic. If q binds X deterministically with the value 2 before p gets a chance to bind it then the deterministic computation in q will be executed only once [39]. Since 2 is the only possible binding for X, p can discard its first and third clauses.

More parallelism can thus be exploited in EDDAP, since as soon as the deterministic consumer instantiates the dependent variable, other suspended non-deterministic goals can resume their execution. In DDAP, these consumer goals would have been suspended until the current leftmost producer goal produces a binding. If none of the goals executed in parallel are deterministic then all bindings of the goals need to be made [39].

That mechanism is further improved by reordering the goals at compile-time. Deterministic goals are moved to the left so that we increase the chances that they will be executed sooner. Another optimization applies to the suspended consumers. Instead of suspending the non-deterministic consumers, *"they are allowed to execute locally so far as they do not influence the computation going on outside of them"* as stated in [39].

When compared to Mercury, the producer/consumer relationships are not detected at run-time but at compile-time thanks to the mode system. However, EDDAP's optimizations are similar to what has been implemented in Mercury for dependent AND-parallelism. Indeed, Peter Wang's modifications reorder the goals so that the producer is the leftmost goal and the consumers are the goals on the right. Doing so reduces the overhead associated with the synchronization mechanisms. Moreover, the *wait* and *signal* calls described in Section 3.2 happen respectively as soon and as late as possible in the call graph so the maximum amount of parallelism is exploited [15].

As said in Section 3, backtracking does not mix well with dependent AND-parallelism as it increases the complexity of a dependent AND-parallelism implementation. Indeed, as stated in [40] : "*whenever a binding to a shared variable is undone, any dependent goal that may have consumed that binding needs to be rolled back to the point at which the consumption took place*". Therefore, dependent AND-parallelism requires more complex roll back patterns than independent AND-parallelism. The two following examples are combinations of producers and consumers which illustrate the mechanisms used for dealing with dependent AND-parallelism and backtracking. The same considerations apply for more complex scenarios [39].

- *p(X)* and *q(X)*, q is a deterministic consumer. When q produces a binding for X, the alternatives in the producer that are incompatible with that binding are pruned, thus, reducing the search space of the producer. If the producer fails with any of the compatible alternatives, then the pruned alternatives would not have helped in any way since they are incompatible with the binding made in q.

- *p(X, Y)* and *q(X, Y)*. Assume that p produces a binding for X making q deterministic. Indeed, the binding prunes alternatives which are incompatible with it, leaving only one which is compatible. In turn, q produces a binding for Y which prunes some alternatives in p. Upon failure p will not be able to see the pruned alternatives. If the binding made by q is the correct one, then p must try another alternative. If the binding made by q is incorrect, then it means that the binding in p which made q deterministic is incorrect.

Since the binding for Y is dependent to the binding for X, the alternatives pruned in q were saved. p will backtrack to a new value for X, which will cause q to take a different alternative.

Teams of processors

Exploiting OR- and AND-parallelism simultaneously gives rise to the following problem. In the context of AND-parallelism, each binding produced by a processor should see the bindings made by the other processors. However, for OR-parallelism, the bindings should remain private among processors. In order to solve that issue, ACE introduces the concept of teams of processors : *"AND-parallelism is exploited between processors within a team while OR-parallelism is exploited between teams"* as stated in [1]. Thus, each team represents an OR-parallel environment and shared data are restricted to happen only among the processors of the same team. Let us have a look at how this applies to C-trees.

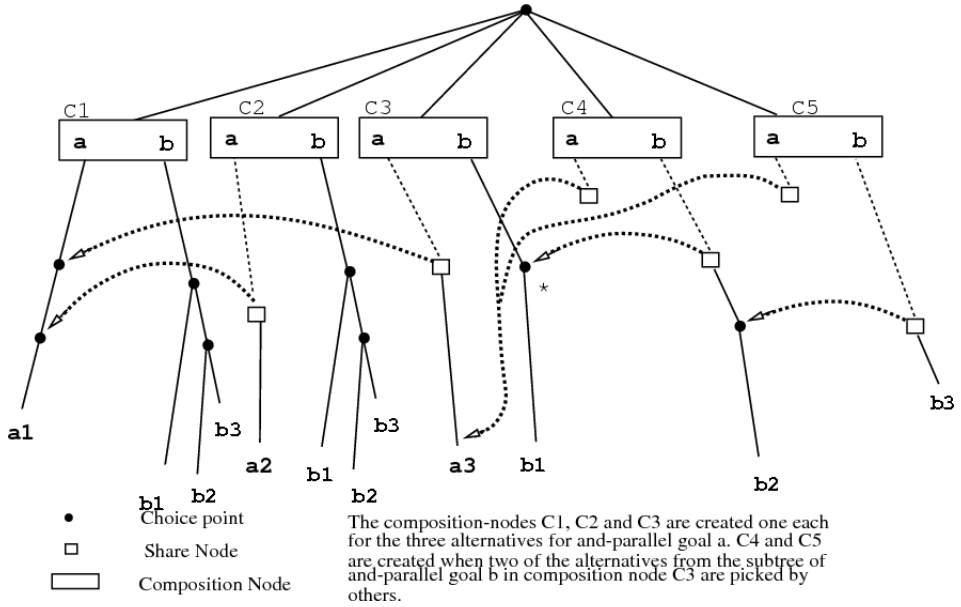


Figure 6.3: C-tree and teams of processors from [1]

Figure 6.3 represents the exploitation of the C-tree on Figure 6.2 by teams of processors. The execution goes as follows. The AND-parallel goals a and b are picked by a team. They produce solutions *a1* and *b1*. In the process, they leave choice points behind. These untried alternatives can be processed in OR-parallel by others teams. The number of OR parallel environments in a C-tree can vary on the number of processors available [1].

Teams of processors can also be exploited in extended AND-OR trees like illustrated on Figure 6.4. On a cross-product node, the processors of a team will work on the different AND-parallel branches of the tree. Whenever a processor finds a solution to a goal, it constructs all of the tuples of the cross-product that can be constructed at that moment. One of these tuples is then exploited for the continuation of the program by another team. Each of these exploited tuples represents an OR-parallel environment [1].

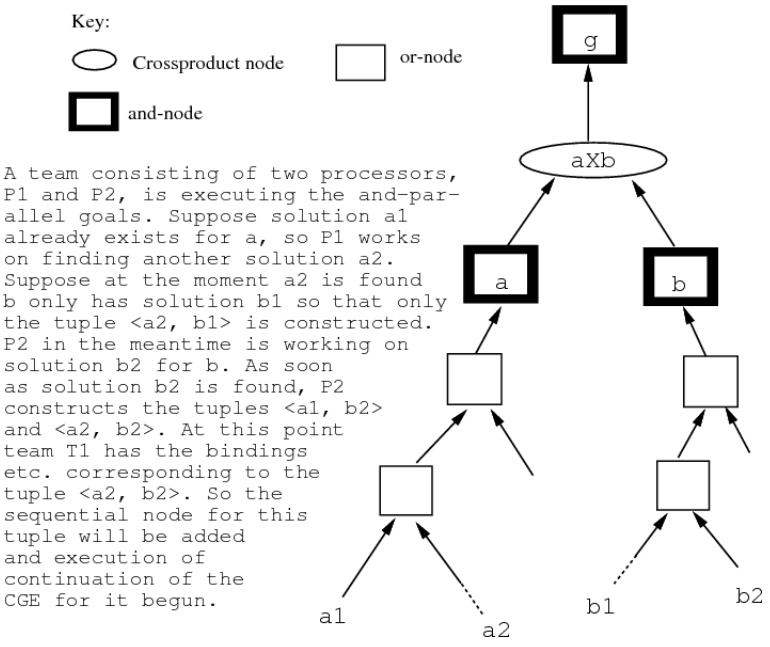


Figure 6.4: Extended AND-OR tree and teams of processors from [1]

Chapter 7

Conclusion

Mercury is undoubtedly a powerful declarative programming language. Its major force is its type, mode and determinism system allowing a large percentage of errors to be detected at compile-time. Therefore, coding in Mercury can be substantially faster than in other languages. The type, mode and determinism system does not only benefit to the end users but also to the developers of the Mercury compiler. Indeed, that system has proved to be a suitable basis for implementing independent parallelism but also dependent parallelism. Compared to Prolog, Mercury's parallelism implementation has been straightforward thanks to the type, mode and determinism system. After having successfully implemented explicit parallelism in Mercury, another complementary approach has been taken to let the compiler decide by itself which goals are to be parallelized. We have also shown how granularity control can improve the performance of Mercury parallel programs, especially recursive predicates which do little work around the recursive calls, whether or not parallelism has been introduced by the user with the explicit parallel operator or through implicit parallelism. However, the current parallelism implementation of Mercury needs various improvements.

First of all, in the current implementation of the implicit parallelism transformation, the N value, deciding whether two gaols are worth parallelizing, has been fixed to 1. That value should be accurately determined such that, on average, only the goals which are worth parallelizing are retained. However, it remains to be seen if this is the most appropriate way of assessing the number of synchronization mechanisms, impacting on the parallel performance, required for two goals to be executed in parallel.

Secondly, the distance granularity control that we have implemented needs to be further improved. The threshold for controlling the granularity of parallelism should be determined by profiling instead of specifying it manually. Moreover, the granularity control should apply to all predicates, and not only to recursive predicates. As a result, static analysis could be implemented and the resulting benefits be compared to the one of the distance granularity control.

Thirdly, the resources available for parallelism should be more taken into account. The run-time granularity control in its current state prevents the creation of a higher number of tasks than the number of available CPUs. However, since a parallel conjunction can contain more than two conjuncts, there might not be a sufficient number of free CPUs available to execute all of the tasks but there might be enough CPUs to execute at least two of the tasks. Therefore, the current run-time granularity control does not try to execute in parallel a maximum number of conjuncts of a parallel conjunction. In order to exploit the maximum

amount of parallelism, the run-time granularity control could be improved such that parallel conjunctions would be transformed to contain as many as parallel conjuncts as the number of free CPU's available at the time the run-time granularity control happens.

Fourthly, besides granularity control, more can be done in order to achieve better parallel performance. As Peter Wang suggested in [15], garbage collection and work balancing are two areas in which improvements can be made and which could result in a performance increase.

As described in [15], the garbage collector in its current implementation has a negative impact on parallel execution. Peter Wang has attributed much of the performance loss in the parallel execution to the garbage collector. Indeed, during the execution of the garbage collector, all program threads are suspended, which is relatively more costy for a parallel program than for a sequential program. Future improvements or a different implementation of the garbage collector need to be made in order to improve the performance of parallel execution of Mercury programs.

As said in Section 6, a work-stealing approach could be implemented into the execution layer. An incomplete work-stealing algorithm is already implemented. As stated in [15], it remains to be see if that approach could result in a performance increase.

Finally, we have seen in ACE that OR-parallelism coupled to AND-parallelism allows to capture the full amount of parallelism available. The Mercury compiler does not implement OR-parallelism since non-determinism is rarely used. However, non-determinism is one of the key features of logic programming and is often seen as indispensable for obtaining more declarative code. Therefore, implementing OR-parallelism could provide a performance increase for non-determinism code. In addition, non-determinism could be more heavily used in the years to come as the Mercury compiler matures as its users community, making thus OR-parallelism valuable. The concept of teams of processors implemented in ACE could be used in Mercury to make AND-OR parallelism coexist.

Bibliography

[1] G. Gupta and V. Santos Costa. Optimal implementation of and-or parallel prolog. In *PARLE'92. Conference, Paris , FRANCE (06/1992)*, volume 10, pages 71–92, 1994. citeseer.ist.psu.edu/304064.htm.

[2] Olof Torgersson. A note on declarative programming paradigms and the future of definitional programming. In *Proceedings of Das Winterte'96*.

[3] Imperative language from foldoc. http://ftp.sunet.se/foldoc/foldoc.cgi?imperative+languages.

[4] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[5] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM (Commun. ACM) ISSN 0001-0782 CODEN CACMA2 Communications of the Association for Computing Machinery*, 31(1):38–43, 1988.

[6] Roy Furman. Declarative strategies for solving software problems, 2005. http://www.sstnet.com/articles_declarative.html.

[7] Sergio Antoy. Functional logic programming. http://web.cecs.pdx.edu/~antoy/research/flp/index.html. Portland State University.

[8] Wim Vanhoof. Programmation fonctionnelle et logique, Academic year 2004-2005. Course at the University of Namur.

[9] Charles K. Nicholas and Timothy W. Finin. Introduction to logic programming and prolog. http://www.cs.umbc.edu/courses/undergraduate/CMSC331/fall00/notes/introlp.pdf, 2000. Course at the University of Maryland, Baltimore County.

[10] Definite clause deduction applet tutorials. http://www.cs.ubc.ca/labs/lci/CIspace/version2/project/CIspace/tree/tut3.html, 2002. Tutorial from the University of British Columbia.

[11] Thomas Conway. *Towards Parallel Mercury*. PhD thesis, University of Melbourne, Australia, Melbourne, Australia, August, 2002.

[12] Ralph Becket. *Mercury tutorial*. University of Melbourne, Australia, Melbourne, Australia, 2006.

[13] Zoltan Somogyi David Jeffery Peter Schachte Simon Taylor Chris Speirs Tyson Dowd Ralph Becket Fergus Henderson, Thomas Conway and Mark Brown. *The Mercury Language Reference Manual*. University of Melbourne, Australia, Melbourne, Australia, 2006.

[14] Simon Taylor. Optimization of mercury programs. Technical report, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia, 1998.

[15] Peter Wang and Zoltan Somogyi. Minimizing the overheads of dependent and-parallelism. Technical report, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia, October, 2006.

[16] Zoltan Somogyi Peter Ross Tyson Dowd Mark Brown Fergus Henderson, Thomas Conway and Ian MacLarty. *The Mercury User's Guide*. University of Melbourne, Australia, Melbourne, Australia, 2006.

[17] Ali E. Abdallah. Reduction strategies and lazy evaluation. `http://myweb.lsbu.ac.uk/~abdallae/units/fp/reduce.pdf`. Course at London South Bank University.

[18] J. Launchbury. Lazy imperative programming. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, DK, SIPL '92*, pages 46–56, 1993. `citeseer.ist.psu.edu/launchbury93lazy.html`.

[19] Yuan Lin. Concurrency vs parallelism, concurrent programming vs parallel programming, June 2006. `http://blogs.sun.com/yuanlin/entry/concurrency_vs_parallelism_concurrent_programming`.

[20] P.J. Plauger and Jim Brodie. Functions. `http://www-ccs.ucsd.edu/c/function.html`, 1996. Manual from the University of California, San Diego.

[21] Philip Fong. Lisp tutorial lecture 4: Imperative programming. `http://www.cs.sfu.ca/CC/310/pwfong/Lisp/4/tutorial4.html`. Tutorial from Simon Fraser University.

[22] Blaise Barney. *Introduction to Parallel Computing*. Livermore Computing, 2006. `http://www.llnl.gov/computing/tutorials/parallel_comp/`.

[23] Vincent W. Freeh. A comparison of implicit and explicit parallel programming. *Journal of Parallel and Distributed Computing*, 34(1):50–65, 1996. `citeseer.ist.psu.edu/freeh94comparison.html`.

[24] Thomas Conway and Zoltan Somogyi. Deep profiling: engineering a profiler for a declarative programming language. Technical report, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia, July, 2001.

[25] P. Lopez, Manuel V. Hermenegildo, and Saumya K. Debray. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation*, 21(4):715–734, 1996. `citeseer.ist.psu.edu/lopez96methodology.html`.

[26] Kish Shen, Vitor Santos Costa, and Andy King. Distance: A new metric for controlling granularity for parallel execution. *Journal of Functional and Logic Programming*, 1999(Special Issue 1), 1999. `citeseer.ist.psu.edu/shen98distance.html`.

[27] Wikipedia. `http://en.wikipedia.org/wiki/`.

[28] Rita Loogen Werner Hans and Stephan Winkler. On the interaction of lazy evaluation and backtracking. Technical report, RWTH Aachen, Lehrstuhl fur Informatik II Ahornstrae 55, W-5100 Aachen, Germany, 1992.

[29] Don Sannella. Lazy and eager evaluation. `http://homepages.inf.ed.ac.uk/dts/fps/lecture-notes/lazy.pdf`, October 2005. Course at the University of Edinburgh.

[30] Theodore Norvell. Monads for the working haskell programmer. `http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm`. Tutorial from Memorial University of Newfoundland.

[31] Mustafa KH. Aswad. Multi-architecture parallel programming using gph, a functional language. `citeseer.ist.psu.edu/aswad02multiarchitecture.html`.

[32] H. Loidl, F. Diez, N. Scaife, K. Hammond, U. Klusik, R. Loogen, G. Michaelson, S. Horiguchi, R. Mari, S. Priebe, A. Portillo, and P. Trinder. Comparing parallel functional languages: Programming and performance, 2002. `citeseer.ist.psu.edu/article/loidl01comparing.html`.

[33] Home page of glasgow parallel haskell. `http://www.macs.hw.ac.uk/~dsg/gph/`.

[34] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *International Workshop on the Implementation of Functional Languages*, Bastad, Sweden, September 1995.

[35] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *The second ACM SIGPLAN conference on History of programming languages*, pages 37–52, November 1992.

[36] Robert Rosebrugh. Prolog: a brief history, 1999. `http://www.mta.ca/~rrosebru/oldcourse/371199/prolog/history.html`.

[37] Home page of cia prolog. `http://www.clip.dia.fi.upm.es/Software/Ciao/`.

[38] Home page of ace: And/or-parallel implementation of prolog. `http://www.cs.nmsu.edu/~gupta/ace.html`.

[39] Enrico Pontelli and Gopal Gupta. Extended dynamic dependent and-parallelism in ACE. *Journal of Functional and Logic Programming*, 1999(Special Issue 1), 1999. `citeseer.ist.psu.edu/pontelli97extended.html`.

[40] E. Pontelli and G. Gupta. Analysis of dependent and-parallelism. In *Fourth CompulogNet Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Languages*, pages 73–91, September 1996.

[41] Introduction to parallel programming. `http://www.mhpcc.edu/training/workshop/parallel_intro/MAIN.html`, 2003. Workshop from Maui High Performance Computing Center.

[42] Robert Harper. Lazy data structures. `http://www.cs.cmu.edu/~rwh/introsml/core/lazydata.htm`, 2000. Notes from Carnegie Mellon University.